STRUCTURED COMPUTER ORGANIZATION

ANDREW S. TANENBAUM

tice-Hall Series in Automatic Computation

STRUCTURED
COMPUTER ORGANIZATION

Prentice-Hall
Series in Automatic Computation

AHO, ed., *Currents in the Theory of Computing*
AHO AND ULLMAN, *The Theory of Parsing, Translation, and Compiling*, Volume I: *Parsing;* Volume II: *Compiling*
ANDREE, *Computer Programming: Techniques, Analysis, and Mathematics*
ANSELONE, *Collectively Compact Operator Approximation Theory and Applications to Integral Equations*
BATES AND DOUGLAS, *Programming Language One*, 2nd ed.
BLUMENTHAL, *Management Information Systems*
BRENT, *Algorithms for Minimization without Derivatives*
BRINCH HANSEN, *Operating System Principles*
COFFMAN AND DENNING, *Operating Systems Theory*
CRESS, et al., *FORTRAN IV with WATFOR and WATFIV*
DAHLQUIST, BJÖRCK, AND ANDERSON, *Numerical Methods*
DANIEL, *The Approximate Minimization of Functionals*
DEO, *Graph Theory with Applications to Engineering and Computer Science*
DESMONDE, *Computers and Their Uses*, 2nd ed.
DRUMMOND, *Evaluation and Measurement Techniques for Digital Computer Systems*
ECKHOUSE, *Minicomputer Systems: Organization and Programming (PDP-11)*
FIKE, *Computer Evaluation of Mathematical Functions*
FIKE, *PL/1 for Scientific Programmers*
FORSYTHE AND MOLER, *Computer Solution of Linear Algebraic Systems*
GEAR, *Numerical Initial Value Problems in Ordinary Differential Equations*
GORDON, *System Simulation*
GRISWOLD, *String and List Processing in SNOBOL4: Techniques and Applications*
HANSEN, *A Table of Series and Products*
HARTMANIS AND STEARNS, *Algebraic Structure Theory of Sequential Machines*
JACOBY, et al., *Iterative Methods for Nonlinear Optimization Problems*
JOHNSON, *System Structure in Data, Programs, and Computers*
KIVIAT, et al., *The SIMSCRIPT II Programming Language*
LAWSON AND HANSON, *Solving Least Squares Problems*
LORIN, *Parallelism in Hardware and Software: Real and Apparent Concurrency*
LOUDEN AND LEDIN, *Programming the IBM 1130*, 2nd ed.
MARTIN, *Computer Data-Base Organization*
MARTIN, *Design of Man-Computer Dialogues*
MARTIN, *Design of Real-Time Computer Systems*
MARTIN, *Future Developments in Telecommunications*
MARTIN, *Programming Real-Time Computing Systems*
MARTIN, *Security, Accuracy, and Privacy in Computer Systems*
MARTIN, *Systems Analysis for Data Transmission*
MARTIN, *Telecommunications and the Computer*

MARTIN, *Teleprocessing Network Organization*
MARTIN AND NORMAN, *The Computerized Society*
MCKEEMAN, et al., *A Compiler Generator*
MEYERS, *Time-Sharing Computation in the Social Sciences*
MINSKY, *Computation: Finite and Infinite Machines*
NIEVERGELT, et al., *Computer Approaches to Mathematical Problems*
PLANE AND MCMILLAN, *Discrete Optimization: Integer Programming and Network Analysis for Management Decisions*
POLIVKA AND PAKIN, *APL: The Language and Its Usage*
PRITSKER AND KIVIAT, *Simulation with GASP II: A FORTRAN-based Simulation Language*
PYLYSHYN, ed., *Perspectives on the Computer Revolution*
RICH, *Internal Sorting Methods Illustrated with PL/1 Programs*
SACKMAN AND CITRENBAUM, eds., *On-Line Planning: Towards Creative Problem-Solving*
SALTON, ed., *The SMART Retrieval System: Experiments in Automatic Document Processing*
SAMMET, *Programming Languages: History and Fundamentals*
SCHAEFER, *A Mathematical Theory of Global Program Optimization*
SCHULTZ, *Spline Analysis*
SCHWARZ, et al., *Numerical Analysis of Symmetric Matrices*
SHAH, *Engineering Simulation Using Small Scientific Computers*
SHAW, *The Logical Design of Operating Systems*
SHERMAN, *Techniques in Computer Programming*
SIMON AND SIKLOSSY, eds., *Representation and Meaning: Experiments with Information Processing Systems*
STERBENZ, *Floating-Point Computation*
STOUTEMYER, *PL/1 Programming for Engineering and Science*
STRANG AND FIX, *An Analysis of the Finite Element Method*
STROUD, *Approximate Calculation of Multiple Integrals*
TANENBAUM, *Structured Computer Organization*
TAVISS, ed., *The Computer Impact*
UHR, *Pattern Recognition, Learning, and Thought: Computer-Programmed Models of Higher Mental Processes*
VAN TASSEL, *Computer Security Management*
VARGA, *Matrix Iterative Analysis*
WAITE, *Implementing Software for Non-Numeric Application*
WILKINSON, *Rounding Errors in Algebraic Processes*
WIRTH, *Systematic Programming: An Introduction*
YEH, ed., *Applied Computation Theory: Analysis, Design, Modeling*

STRUCTURED COMPUTER ORGANIZATION

ANDREW S. TANENBAUM

Vrije Universiteit
Amsterdam, The Netherlands

PRENTICE-HALL, INC.

ENGLEWOOD CLIFFS, NEW JERSEY

Library of Congress Cataloging in Publication Data

TANENBAUM, ANDREW S. (date)
Structured computer organization.

Bibliography
1. Electronic digital computers—Programming.
I. Title.
QA76.6.T38 001.6'42 74-30322
ISBN 0-13-854505-7

© 1976 by Prentice-Hall, Inc., Englewood Cliffs, N. J.

All rights reserved. No part of this book
may be reproduced in any form or by any means
without permission in writing from the publisher.

10 9 8 7 6 5 4 3 2

Printed in the United States of America

PRENTICE-HALL INTERNATIONAL, INC., *London*
PRENTICE-HALL OF AUSTRALIA, PTY. LTD., *Sydney*
PRENTICE-HALL OF CANADA, LTD., *Toronto*
PRENTICE-HALL OF INDIA PRIVATE LIMITED, *New Delhi*
PRENTICE-HALL OF JAPAN, INC., *Tokyo*
PRENTICE-HALL OF SOUTHEAST ASIA (PTE.) LTD., *Singapore*

To Suzanne, Pluis, and Koe

CONTENTS

PREFACE xv

INSTRUCTOR'S PREFACE xviii

1 INTRODUCTION 1

1.1 Languages, Levels, and Virtual Machines 3
1.2 Contemporary Multilevel Machines 4
1.3 Historical Evolution of Multilevel Machines 7
1.4 Hardware, Software, and Multilevel Machines 10
1.5 Processes 12
1.6 Outline of this Book 14

2 COMPUTER SYSTEMS ORGANIZATION 18

2.1 Processors 18
2.1.1 Instruction execution 19
2.1.2 Parallel instruction execution 22
2.2 Memory 24
2.2.1 Bits 25
2.2.2 Memory addresses 25
2.2.3 Metabits 28
2.2.4 Secondary memory 29
Magnetic tapes 29
Magnetic disks 30
Magnetic drums 32
Optical memories 32
2.3 Input/Output 33
2.3.1 I/O devices 33
2.3.2 I/O processors 33
2.3.3 Character codes 34
2.3.4 Error-correcting codes 34
2.3.5 Frequency-dependent codes 38
2.4 Transfer of Information 41
2.4.1 Data paths 41
2.4.2 Telecommunications 42

Modulation 43
Asynchronous and synchronous transmission 45
Simplex, half-duplex, and full-duplex transmission 46
2.5 Computer Networks 47
2.6 Distributed Computers 49

3 THE CONVENTIONAL MACHINE LEVEL 56

3.1 Examples of the Conventional Machine Level 56
3.1.1 IBM System/360 and System/370 57
3.1.2 CDC 6000, Cyber 70, and Cyber 170 62
3.1.3 DEC PDP-11 65
3.2 Instruction Formats 70
3.2.1 Design criteria for instruction formats 71
3.2.2 Expanding opcodes 73
3.2.3 Examples of instruction formats 75
3.3 Addressing 79
3.3.1 Immediate addressing 80
3.3.2 Direct addressing 81
3.3.3 Register addressing 82
3.3.4 Indirect addressing 82
3.3.5 Indexing 84
3.3.6 Base registers 85
3.3.7 Stack addressing 87
Reverse Polish 89
Evaluation of reverse Polish formulas 90
3.3.8 Addressing on the PDP-11 93
3.3.9 Discussion of addressing modes 97
3.4 Instruction Types 98
3.4.1 Data movement instructions 99
3.4.2 Dyadic operations 100
3.4.3 Monadic operations 102
3.4.4 Comparisons and conditional jumps 104
3.4.5 Procedure call instructions 106
3.4.6 Loop control 107
3.4.7 Input/output 108
3.5 Representation of Data 111
3.5.1 Integers 112
3.5.2 Floating-point numbers 112
3.5.3 Booleans 112
3.5.4 Characters 113
3.5.5 Strings 113
3.5.6 Arrays 116
Dope vectors 116
Marginal indexing 117
3.6 Flow of Control 119
3.6.1 Sequential flow of control and jumps 119
3.6.2 Procedures 120
3.6.3 Coroutines 125
3.6.4 Traps 130
3.6.5 Interrupts 131

4 THE MICROPROGRAMMING LEVEL 141

4.1 Processor Components 142
4.1.1 Registers 142
4.1.2 Buses 143
4.1.3 Gates 143
4.1.4 Clocks 144
4.1.5 Memory ports 145
4.1.6 Arithmetic and logical units 146
4.1.7 Packaging of processor components 147
4.2 Basic Operations 148
4.2.1 Register transfer 148
4.2.2 Memory read/write 150
4.2.3 Bit testing 150
4.3 A Hypothetical Target Level 150
4.4 A Hypothetical Host Level 153
4.4.1 The host level's registers 153
4.4.2 The host level's ALU 155
4.4.3 The host level's gates and data paths 155
4.5 Gate Sequences 157
4.5.1 Subcycles 157
4.5.2 Gate sequences for the ADD instruction 158
4.6 Microprogrammed Gate Control 160
4.6.1 Microinstructions 161
4.6.2 Execution of microprograms 162
4.6.3 A two-level machine 164
4.7 A Language for Microprogramming 165
4.7.1 Notation for GATE microinstructions 165
4.7.2 Notation for TEST microinstructions 166
4.8 The Interpreter for the Target Machine 167
4.8.1 Interpretation of the multiplication instruction 171
4.8.2 Interpretation of the division instruction 172
4.8.3 Perspective 175
4.9 Design of the Microprogramming Level 175
4.9.1 Encoded fields 176
4.9.2 Horizontal versus vertical organization 177
4.9.3 Memory cycle ratios and overlapped execution 180
4.9.4 Nanomemories 183
4.9.5 Universal versus specific microprogramming levels 185
4.9.6 Review of microprogramming level organization 186
4.10 Advantages and Disadvantages of Microprogramming 188
4.11 The IBM 370/125's Microprogramming Level 190
4.11.1 Architecture of the IBM 370/125 microprogramming level 190
4.11.2 IBM 3125 microinstructions 194
4.12 The PDP-11/40 microprogramming level 196
4.12.1 Architecture of the PDP-11/40 microprogramming level 196
4.12.2 UNIBUS operation 199
4.12.3 PDP-11/40 microinstructions 201
4.13 The Burroughs B1700 204
4.13.1 Architecture of the B1700 206
4.13.2 The B1700 instruction set 209

5 THE OPERATING SYSTEM MACHINE LEVEL 215

5.1 Implementation of the Operating System Machine Level 215
5.2 Virtual I/O Instructions 218
5.2.1 Sequential files 219
5.2.2 Random access files 220
5.2.3 Implementation of virtual i/o instructions 221
5.2.4 IBM 370 virtual i/o 226
5.2.5 Cyber 70 virtual i/o 229
5.2.6 PDP-11 virtual i/o 235
5.2.7 Comparison of level 3 i/o 236
5.3 Virtual Instructions Used in Parallel Processing 237
5.3.1 Process creation and destruction 238
5.3.2 Race conditions 239
5.3.3 Process synchronization using semaphores 242
5.3.4 Instructions for interprocess communication 245
5.4 Other Level 3 Instructions 246
5.4.1 Directory management instructions 246
5.4.2 Reconfiguring the level 3 machine 247
5.5 Virtual Memory 249
5.5.1 Paging 250
5.5.2 Implementation of paging 252
5.5.3 Demand paging and the working set model 256
5.5.4 Page replacement policy 259
5.5.5 The dirty bit 261
5.5.6 The hardware map 261
5.5.7 Page size and fragmentation 262
5.5.8 Cache memory 263
5.5.9 Segmentation 264
5.5.10 Virtual memory on the PDP-11 268
Checkerboarding 271
5.5.11 The MULTICS virtual memory 273
5.5.12 Virtual memory on the IBM 370 277
5.5.13 Segmented virtual memory and file i/o 278
5.6 Job Control Languages 279

6 THE ASSEMBLY LANGUAGE LEVEL 288

6.1 Introduction to Assembly Language 289
6.1.1 What is an assembly language? 289
6.1.2 Format of an assembly language statement 290
6.1.3 Comparison of assembly language and PL/I 293
6.1.4 Program tuning 293
6.2 The Assembly Process 296
6.2.1 Two-pass assemblers 296
6.2.2 Pass one 297
6.2.3 Pass two 301
6.3 Searching and Sorting 303
6.3.1 Searching 303
6.3.2 Linear searching 304

6.3.3 Binary searching 305
6.3.4 Sorting 307
6.3.5 Hash coding 308
6.3.6 Hashing functions and collisions 312
6.3.7 Comparison of association techniques 315
6.4 Macros 316
6.4.1 Macro definition, call, and expansion 316
6.4.2 Macros with parameters 319
6.4.3 Conditional macro expansion 320
6.4.4 Nested macro calls 322
6.4.5 Recursive macro calls 323
6.4.6 Nested macro definitions 325
6.4.7 Implementation of a macro facility in an assembler 326
6.5 Linking and Loading 328
6.5.1 Tasks performed by the linker 329
6.5.2 Structure of an object module 332
6.5.3 Binding time and dynamic relocation 333
6.5.4 Dynamic linking 336

7 MULTILEVEL MACHINES 344

7.1 Methods of Implementing New Levels 344
7.1.1 Interpretation 344
7.1.2 Translation 346
General-purpose macro processors 347
7.1.3 Procedural extension 349
7.2 Design Strategies for Multilevel Machines 349
7.2.1 Top-down design 350
7.2.2 Bottom-up design 352
7.2.3 Middle-out design 354
7.3 Program Portability 354
7.3.1 A universal programming language 355
7.3.2 The brute force approach 356
7.3.3 UNCOL 357
7.3.4 Do-it-yourself virtual machines 360
7.3.5 Emulation 363
7.3.6 Networks 364
7.4 Self-Virtualizing Machines 364
7.4.1 IBM VM/370 system 365
7.4.2 Goals of self-virtualizing machines 367
Self-virtualizing machines and time sharing 368
Operating system testing 368
Protection of confidental data 369
7.4.3 Implementation of a self-virtualizing machine 370
Exceptions and virtual machine faults 370
Simulation of virtual machine i/o 372
Self-modifying channel programs 372
Shadow page tables 373
7.5 High-Level Machine Architecture 375
7.5.1 Addressing and descriptors 377
7.5.2 High-level machine instructions 381
7.5.3 Advantages and disadvantages of high-level machines 383

8 SUGGESTIONS FOR FURTHER READING AND BIBLIOGRAPHY 387

8.1 Suggestions for Further Reading 387
8.1.1 Addressing and instructions 387
8.1.2 Assemblers and assembly language programming 388
8.1.3 Binary numbers and arithmetic 389
8.1.4 Character codes, redundant and nonredundant 390
8.1.5 Computer organization 390
8.1.6 The conventional machine level 391
8.1.7 Deadlocks 393
8.1.8 File systems 393
8.1.9 High-level machines 394
8.1.10 Input/output 395
8.1.11 Linkers and loaders 395
8.1.12 Macros 396
8.1.13 The microprogramming level 397
8.1.14 Multilevel computers 398
8.1.15 Networks 399
8.1.16 Operating systems 400
8.1.17 Parallel processing 401
8.1.18 Self-virtualizing machines 402
8.1.19 Symbol tables 403
8.1.20 Telecommunications 405
8.1.21 Virtual memory 405
8.2 Alphabetical Bibliography 407

APPENDIX

A FINITE-PRECISION ARITHMETIC AND BINARY NUMBERS 413

A.1 Finite-Precision Numbers 413
A.2 Radix Number Systems 415
A.3 Conversion from One Radix to Another 416
A.4 Negative Binary Numbers 420
A.5 Binary Arithmetic 421

APPENDIX

B FLOATING-POINT NUMBERS 424

APPENDIX

C BOOLEAN ALGEBRA 432

INDEX 436

PREFACE

Once upon a time computers were very simple. These early machines executed a small number of elementary instructions, and the user wrote his programs directly using these primitive instructions. Those days are long gone. A modern computer is a far more complicated entity, often consisting of a half dozen or more distinct levels at which it can be programmed and studied. In fact, nowadays it is often difficult to tell where the "machine" ends and where the software begins.

Many universities have a course in assembly language programming and computer organization early in the curriculum. There was a time when teaching the students how to program one specific computer in assembly language was considered adequate. Those days are also long gone. These courses must keep pace with a rapidly changing subject, and must now provide an introduction to a wide range of topics related to computer organization, many of which were barely known outside the research laboratories only a few years ago. Segmented virtual memories, parallel processing, race conditions, microprogramming, variable architecture machines, store and forward networks, self-virtualizing machines, and cache memories are just a few examples of these varied topics.

This book is intended as a textbook for such an introductory course on assembly language programming and computer organization. The only prerequisite is an introductory course in computer science (or equivalent practical experience) including a little knowledge of programming in a high-level language such as FORTRAN, COBOL, or PL/I. No mathematical or engineering background is needed, making the book suitable for use even at the sophomore level. Nearly all of the material of the B2 course in the ACM Curriculum 68 is covered, as well as similar material that has become important since that proposal was written. The book can also be used for the proposed COSINE course on "Machine Structure and Machine Language Programming." The chapters are self-contained, allowing them to be used as references on specific subjects.

The structured organization of computers, as a hierarchy of levels, is the theme of this book. At the bottom level is the true hardware, whose function is to execute

interpreters called microprograms. The language interpreted by the microprograms is the one most people think of as the "machine language" and is the one described in the manufacturer's machine reference manual. Thus, this "machine language" is already one level removed from the language directly executed by the electronic circuits. We call the lowest level "the microprogramming level" (Chapter 4) and the level it supports "the conventional machine level" (Chapter 3).

Most computers have an operating system that runs on the conventional machine level. The operating system provides its users with an "extended" or "virtual" machine that has instructions and facilities not present at the conventional machine level. These include file manipulation instructions, instructions for parallel processing (multitasking) and virtual memory. The set of features and instructions available to the user of the operating system can be regarded as defining a new level, "the operating system machine level" (Chapter 5).

The fourth level is the symbolic assembly language level. Unlike levels 2 and 3, which are supported by interpretation, this level is implemented by a program called an assembler (Chapter 6), which translates level 4 programs to one of the levels below it. Still higher levels, not covered in this book, are the levels defined by problem-oriented languages.

Chapter 7 examines the construction and applications of multilevel computers as a whole, in contrast to Chapters 3–6, each of which deals only with a single level. Chapter 8 is a guide to further study.

The IBM 370, CDC Cyber 70, and DEC PDP-11 computers are used as running examples throughout the text. These machines are used for illustrative purposes only, and no prior experience with any of them is expected.

There are a few algorithms written in a simple subset of PL/I, but these should be immediately understandable to anyone familiar with FORTRAN or ALGOL. PL/I was chosen as a compromise, since it is both widely used and suitable for structured programming. My own first choice was ALGOL 68, but few sophomores know it, unfortunately.

Many people have made contributions to this book. At times I have felt more like an editor than an author. Two people stand out above the others. Jack Alanen read the first handwritten draft in its entirety, and kept many bad ideas from even getting as far as the typewriter. He also made numerous suggestions for improving both the content and presentation. I especially thank Jack for teaching me a lot about teaching. Kim Gostelow went over a later version with a fine-tooth comb, commenting extensively, correcting errors, and trying to turn my prose into something resembling English. Nearly every single paragraph in the entire manuscript was marked with some suggestion for improvement. I am deeply indebted to both of them.

Reind van de Riet and Mitchell Tanenbaum also read and criticized the complete manuscript, providing me with different points of view. John W. Carr III, Arnie Falick, Dick Grune, Jim van Keulen, Bob Rosin, Carel Stillebroer and Wayne Wilner each offered suggestions and help with specific sections. I would also like to thank my students, especially Arie de Bruin, Wim Harmsen, Ad König, Sape Mullender, Johan

Stevenson, and Hans van Vliet for providing feedback. Mrs. Homburg-Knieper did a fine job of typing several versions of the manuscript.

I would like to express my appreciation to the International Business Machines Corporation, the Control Data Corporation, the Digital Equipment Corporation and the Burroughs Corporation for permission to adapt material from their copyrighted publications as follows: IBM—"IBM System/370 Principles of Operation Manual" and engineering drawings for the IBM 3125 CPU; CDC—"Control Data Cyber 70 Models 72, 73, 74 Computer Systems Reference Manual"; DEC—"PDP-11/45 Processor Handbook" and PDP-11/40 engineering drawings; Burroughs—"B1700 Systems Reference Manual." Any errors in the descriptions of these computers are my responsibility.

Finally, I want to thank Suzanne for encouragement, support, help, and especially patience throughout my long period of self-imposed exile from the human race during the preparation of this book. I also would like to thank her for breaking up the long hours with an *uitgeperste sinaasappel.*

ANDREW S. TANENBAUM

INSTRUCTOR'S PREFACE

Assembly language programming exercises should be a major part of any course using this book. On the very first day of class the students should be given a thoroughly documented listing of a debugged assembly language program for the computer to be used, along with a brief description of the computer and a handout explaining the administrative details of running programs—where to find terminals or keypunches, what job control cards are needed, how to use the editor, etc. The students are to punch and run the program, so that no time will be wasted later on mechanical details.

The next step is to have them make some trivial change to the program. For example, a cryptography program that encodes or decodes messages using a simple substitution cipher could be modified to handle numbers as well as letters. After making several steadily more difficult changes to one or two simple programs, the students should be ready for a more serious program.

One good candidate involves the instructor providing an assembly language version of the interpreter of Fig. 4-15. The students would be expected to improve the target machine supported by the interpreter. A few suggestions are given in the problems at the end of Chapter 4, but the students should be encouraged to think of ideas themselves. This exercise makes it clear that the "conventional machine level" is really defined by the underlying interpreter (microprogram) and not by the hardware.

Other possible exercises would be simulations of the performance of a computer network or a virtual memory, or modification of a simple assembler. Providing the students with working programs as a basis is preferable to having them write everything from scratch. This spares the slower students from wasting too much time on questions like "In which column do you put the opcode?" At the same time it provides the better students with an easy way to get into more ambitious projects, such as adding a macro facility to an assembler.

Hands-on experience with a minicomputer is preferable to the use of time sharing terminals, which is in turn preferable to a batch environment, but the choice is usually dictated by the availability of equipment. The students should be given specific

reading assignments in the appropriate manufacturer's manuals as early as possible in the course.

This book can either be used for an intensive one semester course or, with a few outside readings, a leisurely one year course. Alternately, a one semester course could concentrate on certain topics and omit other topics. For example, a course on computer organization could use Chapters 1, 2, parts of 3, and then 4 and 7. A course emphasizing assembly language programming could use Chapters 1, 2, 3, and 6. Still another approach is to cover all the chapters, but leave some of the sections containing slightly more advanced ideas or additional examples to the more ambitious students.

1 INTRODUCTION

A digital computer is a machine that can solve problems for people by carrying out instructions given to it. A sequence of instructions describing how to perform a certain task is called a program. The electronic circuits of each computer can recognize and directly execute a limited set of simple instructions into which all its programs must be converted before they can be executed. These basic instructions are rarely much more complicated than

add two numbers

check a number to see if it is zero

move a piece of data from one part of the computer's memory to another

Together, a computer's primitive instructions form a language in which it is possible for people to communicate with the computer. Such a language is called a **machine language.**

The people designing a new computer must decide what instructions to include in its machine language. Usually they try to make the primitive instructions as simple as possible, consistent with the computer's intended use and performance requirements, in order to reduce the complexity and cost of the electronics needed. Because most machine languages are so simple, it is difficult and tedious for people to use them.

There are two principal ways to attack this problem; both involve designing a new set of instructions that is more convenient for people to use than the set of built-in machine instructions. Taken together, these new instructions also form a

language, which we will call L2, just as the built-in machine instructions form a language, which we will call L1. The two approaches differ in the way programs written in L2 are executed by the computer, which, after all, can only execute programs written in its machine language, L1.

One method of executing a program written in L2 is first to replace each instruction in it by an equivalent sequence of instructions in L1. The result is to yield a new program consisting entirely of L1 instructions. The computer then executes the new L1 program instead of the old L2 program. This technique is called **translation.**

The other technique is to write a program in L1 that takes programs in L2 as input data and carries them out by examining each instruction in turn and executing the equivalent sequence of L1 instructions directly. This technique does not require first generating a new program in L1. It is called **interpretation,** and the program that carries it out is called an **interpreter.**

Translation and interpretation are actually quite similar. In both methods, instructions in L2 are ultimately carried out by executing equivalent sequences of instructions in L1. The difference is that, in translation, the entire L2 program is first converted to an L1 program, the L2 program is thrown away, and then the newly generated L1 program is executed. In interpretation, after each L2 instruction is examined and decoded, it is carried out immediately. No translated program is generated. Both methods are widely used.

Rather than thinking in terms of translation or interpretation, it is often more convenient to imagine the existence of a hypothetical computer or **virtual machine** whose machine language is L2. If such a machine could be constructed cheaply enough, there would be no need for having L1 or a machine that executed programs in L1 at all. People could simply write their programs in L2 and have the computer execute them directly. Even though the virtual machine whose language is L2 is too expensive to construct out of electronic circuits, people can write programs for it and have these programs either interpreted or translated by a program written in L1 that can be directly executed by the existing computer. In other words, people can write programs for virtual machines, just as though they really existed.

In order to make translation or interpretation practical, the languages L1 and L2 must not be "too" different. This constraint often means that L2, although better than L1, will still be far from ideal for most applications. That L2 should be far from ideal is perhaps discouraging in light of the original purpose for creating it—namely, to relieve the programmer of the burden of having to express himself in a language more suited to machines than people. However, the situation is far from hopeless.

One approach is to invent still another set of instructions that are more people oriented and less machine oriented than those of L2. This third set also forms a language, which we will call L3. People can write programs in L3 just as though a virtual machine with L3 as its machine language really existed. Such programs can either be translated to L2 or executed by an interpreter written in L2.

The invention of a whole series of languages, each one more convenient than its predecessors, can go on indefinitely until a suitable one is finally achieved. It is

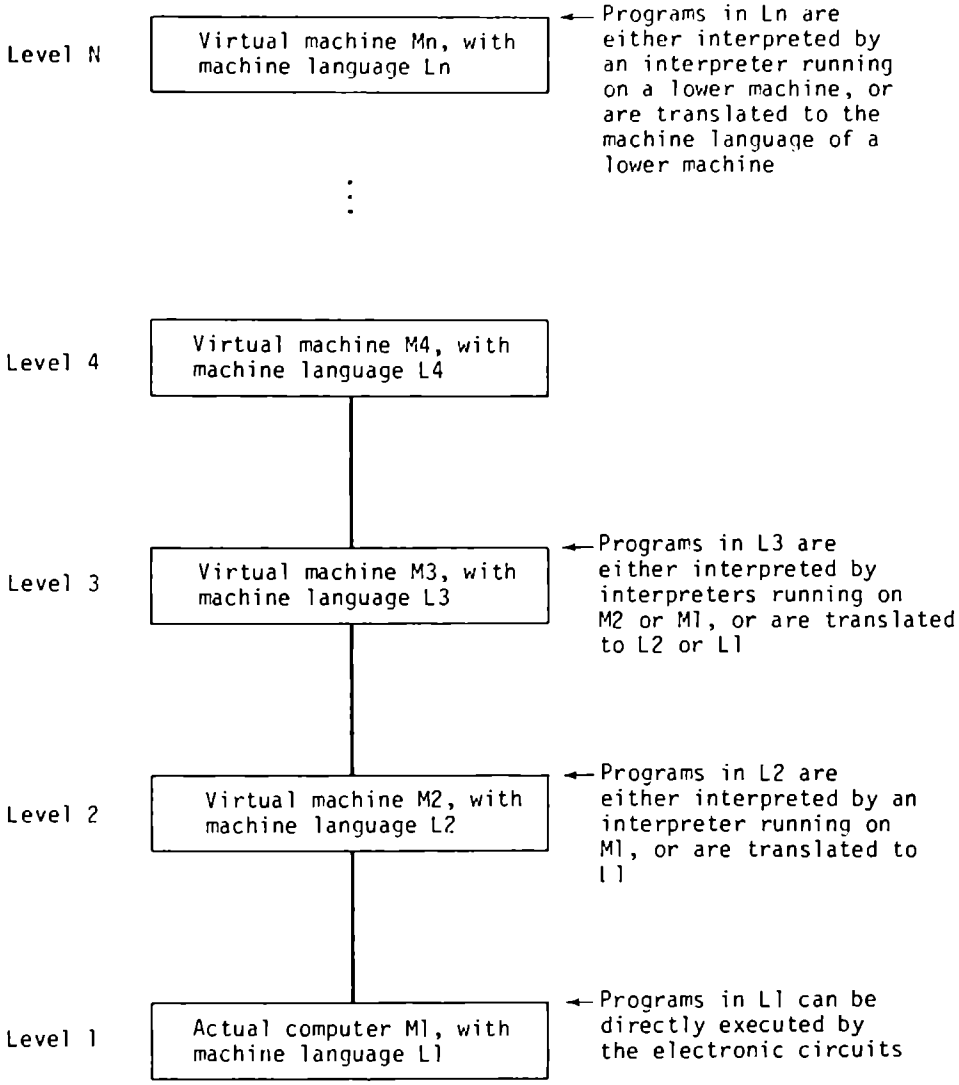

Level N
Virtual machine Mn, with machine language Ln
Programs in Ln are either interpreted by an interpreter running on a lower machine, or are translated to the machine language of a lower machine
Level 4
Virtual machine M4, with machine language L4
Level 3
Virtual machine M3, with machine language L3
Programs in L3 are either interpreted by interpreters running on M2 or M1, or are translated to L2 or L1
Level 2
Virtual machine M2, with machine language L2
Programs in L2 are either interpreted by an interpreter running on M1, or are translated to L1
Level 1
Actual computer M1, with machine language L1
Programs in L1 can be directly executed by the electronic circuits

Fig. 1-1. A multilevel machine.

important to note that each language uses its predecessor as a basis, so we may view a computer using this technique as a series of layers or **levels**, one on top of another, as shown in Fig. 1-1. The bottom-most language or level is the simplest, and the highest language or level is the most sophisticated.

1.1. LANGUAGES, LEVELS, AND VIRTUAL MACHINES

There is an important relation between a language and a virtual machine. Each machine has some machine language, consisting of all the instructions that the machine can execute. In effect, a machine defines a language. Similarly, a language

defines a machine—namely, the machine that can execute all programs written in that language. Of course, the machine defined by a certain language may be enormously complicated and expensive to construct directly out of electronic circuits, but we can imagine it nevertheless. A machine with PL/I, ALGOL 68, or COBOL as its machine language would be a very complex beast indeed, but it is certainly conceivable, and perhaps in a few years such a machine will be considered trivial to build.

A computer with N levels can be regarded as N different virtual machines, each with a different machine language. We will use the terms "level" and "virtual machine" interchangeably. Only programs written in language L1 can be directly carried out by the electronic circuits, without the need for intervening translation or interpretation. Programs written in L2, L3, . . . , Ln must either be interpreted by an interpreter running on a lower level or translated to another language corresponding to a lower level.

A person whose job it is to write programs for the level N virtual machine need not be aware of the underlying interpreters and translators. He knows that he can write programs for the level N virtual machine and that they will somehow be executed. It is of little interest to him whether they are carried out step by step by an interpreter, which, in turn, is also carried out by another interpreter, or whether they are carried out directly by the electronics. He sees the same result in both cases: his programs are executed.

The vast majority of programmers using an N level machine are only interested in the top level, the one least resembling the machine language at the very bottom. However, people interested in understanding how a computer really works must study all the levels. People interested in designing new computers or designing new levels (i.e., new virtual machines) must also be familiar with levels other than the top one. The concepts and techniques of constructing machines as a series of levels and the details of some important levels themselves form the main subject of this book. The title comes from the fact that viewing a computer as a hierarchy of levels provides a good structure or framework for understanding how computers are organized. Furthermore, designing a computer system as a series of levels helps to ensure that the resulting product will be well structured.

1.2. CONTEMPORARY MULTILEVEL MACHINES

Most modern computers consist of two or more levels. Five-level systems are not at all unusual, as shown in Fig. 1-2. The bottom levels of many commercially available computers have much in common in terms of instruction set and general organization. Strictly speaking, two computers with different machine languages have different bottom levels, but enough similarities exist to allow us to abstract the essential features of such a level and discuss it as though it were well defined. For example, few of these levels have more than 20 or 30 distinct kinds of instructions. Most of

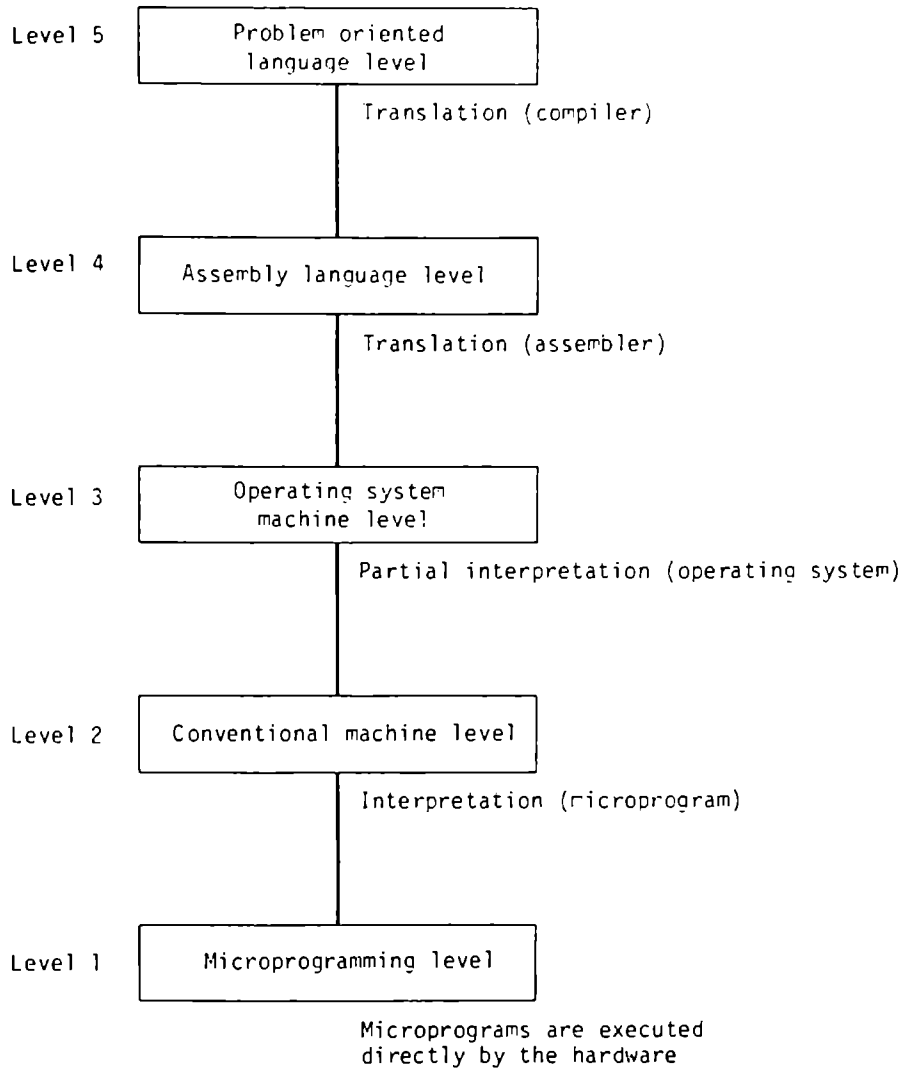


Fig. 1-2. Five levels present on most modern computers. The method by which each level is supported is indicated below it, along with the name of the supporting program in parentheses.

the instructions involve moving data from one part of the computer to another, and a few involve simple tests. This level is called the **microprogramming level,** and the interpreters it executes are called **microprograms.** We will also refer to it as level 1, to emphasize its position as the bottom level on which all the other levels are built. The instructions at this level are carried out by the computer's electronic circuits.

Each level 1 machine has one or more interpreters that can run on it. Each interpreter defines a level 2 language (and a virtual machine, whose machine language is that language). These level 2 machines also have much in common. Even level 2

machines from different manufacturers have more similarities than differences. In this book we will call this level the **conventional machine level,** for lack of a generally agreed upon name.

Each computer manufacturer publishes a manual for each of the computers it sells, entitled "Machine Language Reference Manual" or "Principles of Operation of the Western Wombat Model 100X Computer" or something similar. These manuals are really about the level 2 virtual machine, not the level 1 actual machine. When they describe the machine's instruction set, they are, in fact, describing the instructions carried out interpretively by the microprogram, not the hardware instructions themselves. If a computer manufacturer provided two interpreters for one of its machines, interpreting two different level 2 machine languages, it would need to provide two "machine language" reference manuals, one for each interpreter.

It should be mentioned that some computers, particularly older ones, do not have a microprogramming level. On these machines, the conventional machine level instructions are carried out directly by the electronic circuits, without any intervening interpreter. As a result, level 1, and not level 2, is the conventional machine level. Nevertheless, we will continue to call the conventional machine level "level 2," despite these exceptions.

The third level is usually a hybrid level. Most of the instructions in its language are also in the level 2 language. (There is no reason why an instruction appearing at one level cannot be present at other levels as well.) In addition, there are a set of new instructions, a different memory organization, the ability to run two or more programs in parallel, and various other features. There is more variation between level 3 machines than between either level 1 machines or level 2 machines.

The new facilities added at level 3 are carried out by an interpreter running at level 2, which, historically, has been called an **operating system.** Those level 3 instructions identical to level 2's are carried out directly by the microprogram rather than by the operating system. In other words, some of the level 3 instructions are interpreted by the operating system, and some of the level 3 instructions are interpreted directly by the microprogram. This is what we mean by "hybrid." We will call this level the **operating system machine level.**

There is a fundamental break between levels 3 and 4. The lowest three levels are not actually designed for use by the average garden-variety programmer in solving his problems. They are intended primarily for running the interpreters and translators needed to support the higher levels. These interpreters and translators are written by people called **systems programmers,** who specialize in designing and implementing new virtual machines. Levels 4 and above are intended for the applications programmer with a problem to solve.

Another change occurring at level 4 is the method by which the higher levels are supported. Levels 2 and 3 are always interpreted. Levels 4, 5, and above are usually, although not always, supported by translation.

Yet another difference between levels 1, 2, and 3, on the one hand, and levels 4, 5, and higher on the other is the nature of the language provided. The machine

languages of levels 1, 2, and 3 are numeric. Programs in them consist of long series of numbers, which are great for machines but bad for people. Starting at level 4, the languages contain words and abbreviations meaningful to people.

Level 4, the assembly language level, is really a symbolic form for one of the underlying languages. This level provides a method for people to write programs for level 1, 2, or 3 in a form that is not as unpleasant as the virtual machine languages themselves. Programs in assembly language are first translated to level 1, 2, or 3 language and then interpreted by the appropriate virtual or actual machine. The program that performs the translation is called an **assembler.** Assembly language once was very important, but it is becoming less important as time goes on.

Level 5 consists of languages designed to be used by applications programmers with problems to solve. Such languages are called by many names, including **high-level languages** and **problem-oriented languages.** Literally hundreds of different ones exist. A few of the better known ones are ALGOL 60, ALGOL 68, APL, BASIC, COBOL, FORTRAN, LISP, PASCAL, RPG, and SNOBOL4. Programs written in these languages are generally translated to level 3 or level 4 by translators known as **compilers,** although occasionally they are interpreted instead.

Level 6 and above consist of collections of programs designed to provide machines specifically tailored to certain applications, and containing large amounts of information about that application. It is possible to imagine virtual machines intended for applications in administration, education, computer design, and so on. These levels are an area of current research.

1.3. HISTORICAL EVOLUTION OF MULTILEVEL MACHINES

In order to provide some perspective and insight into multilevel machines, we will briefly examine their historical development. The very first digital computers, back in the 1940s, had only one level, similar to the present conventional machine level. All their programs were directly written in machine language as long series of numbers and were entered into the machine by using switches and wires. The machine instructions were directly executed by the machines' (vacuum tube) circuits.

These circuits were complicated, difficult to understand and build, and not very reliable. In 1951 M. V. Wilkes in England suggested the idea of designing a two-level computer. This machine was to have a built-in, unchangeable interpreter, whose function was to execute conventional machine language programs interpretively. A few of these two-level machines were constructed during the 1950s. More were constructed during the 1960s. By 1970 the idea of having the conventional machine level be interpreted by a microprogram was widespread.

Assemblers and compilers were developed during the 1950s to ease the programmer's task. In those days most computers were "open shop," which meant that the programmer had to operate the machine himself. Next to each machine was a sign-up sheet. If a programmer wanted to run a program, he signed up for a block of time,

say Wednesday morning 3 to 5 A.M. (many programmers liked to work when it was quiet in the machine room). When his time arrived, he headed for the machine room with a deck of cards in one hand and a sharpened pencil in the other. Upon arriving in the computer room, he gently nudged the previous programmer toward the door and took over the computer.

If he wanted to run a FORTRAN program, he went through the following steps:

1. He went over to the cabinet where the program library was kept, took out the big green deck labeled FORTRAN compiler, put it in the card reader, and pushed the start button.
2. He put his FORTRAN program in the card reader and pushed the continue button. The program was read in.
3. When the computer stopped, he read his FORTRAN program in a second time. Although some compilers only required one pass over the input, others required three or more.
4. Finally, the translation neared completion. The programmer often became nervous near the end because if the compiler found an error in his program, he had to correct it and start all over again. If there were no errors, the compiler punched out the translated machine language program on cards.
5. The programmer then put the machine language program in the card reader and read it in.
6. The program began executing. More often than not, it did not work and unexpectedly stopped in the middle. Generally the programmer fiddled with the console switches and looked at the console lights for a little while. If lucky, he figured out the problem, corrected the error, and went back to the cabinet containing the big green FORTRAN compiler to start all over again. If less fortunate, he made a printout of memory and took it home to study.

This system, with minor variations, was normal at many computer centers for years. It forced the programmers to learn how to operate the machine and to know what to do when it broke down, which was often. The machine was frequently idle while people were carrying cards around the room or scratching their heads trying to find out why their program was not behaving the way it was supposed to.

Around 1960 people tried to reduce the amount of wasted time, by automating the operator's job. A program called an operating system was kept in the computer at all times. The programmer provided certain control cards along with the program that were read and carried out by the operating system. Figure 1-3 shows a sample deck for one of the first widespread operating systems, FMS (FORTRAN Monitor System), on the IBM 709.

The operating system read the *JOB card and used the information on it for accounting purposes. Later, it read the *FORTRAN card, which was an instruction to load the FORTRAN compiler from a magnetic tape. When the compiler finished,

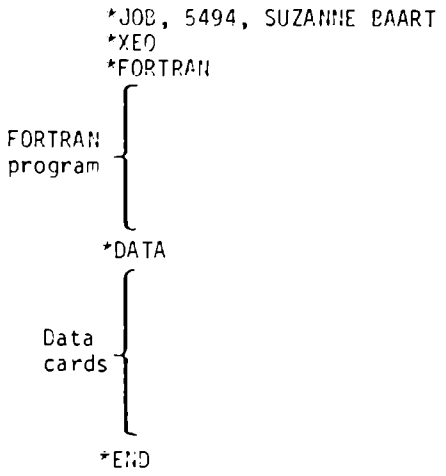


Fig. 1-3. A sample job for the FMS operating system.

it returned control back to the operating system, which then read the *DATA card. This was an instruction to execute the translated program (using the cards following the *DATA card as the data).

It is interesting to note that although the operating system was designed to automate the operator's job (hence the name), it was also the first step in the development of a new virtual machine. The *FORTRAN card could be viewed as a virtual "compile program" instruction. Similarly, the *DATA card could be regarded as a virtual "execute program" instruction. A level with only two instructions was not much of a level, but it was a start in that direction.

In subsequent years, operating systems became more sophisticated. New instructions and features were added to the conventional machine level until it began to take on the appearance of a new level. Some of this new level's instructions were identical to the conventional machine level instructions, but others, particularly input/output instructions, were completely different. The new instructions were often known as "operating system macros" or "supervisor calls," and the terms still linger on.

Operating systems developed in other ways as well. The early ones read card decks and printed output on the line printer. This was known as a **batch** system. There was usually a wait of several hours between the time a program was submitted and the time the results were ready.

In the early 1960s researchers at Dartmouth College, MIT, and elsewhere developed operating systems that allowed programmers to communicate directly with the computer. In these systems, remote terminals were connected to the central computer via telephone lines. A programmer could type in his program and get the results typed back almost immediately, right in his own office, or in his garage at home, or wherever he kept his terminal. These systems were, and still are, called **time-sharing** systems.

Our interest in operating systems is in those parts that interpret the instructions and features present in level 3 and not present in level 2, rather than in the time-

sharing aspects. Although we will not emphasize it, you should keep in mind that operating systems do more than just interpret parts of level 3 programs.

1.4. HARDWARE, SOFTWARE, AND MULTILEVEL MACHINES

Programs written in a computer's machine language can be directly executed by the computer's electronic circuits, without the need for any intervening interpreters or translators. These electronic circuits, along with the memory and input/output devices, form the computer's **hardware.** Hardware consists of tangible objects—integrated circuits, printed circuit boards, cables, power supplies, memories, card readers, line printers, and terminals—rather than abstract ideas or instructions. How these components are constructed and how they work belong to the domain of electrical engineering.

Software, in contrast, consists of algorithms (detailed instructions telling how to do something) and their computer representations—namely, programs. Programs can be represented on punched cards, magnetic tape, photographic film, and other media, but the essence of software is the set of instructions that make up the programs, not the physical media on which they are recorded.

A central theme of this book that will occur over and over again is: *hardware and software are logically equivalent.*

Any operation performed by software can also be built directly into the hardware, and any instruction executed by the hardware can also be simulated in software. The decision to put certain functions in the hardware and others in the software is made on the basis of such factors as cost, speed, amount of memory required, reliability, and frequency of expected changes. There are no hard and fast rules to the effect that X must go into the hardware and Y must be programmed explicitly. Designers with different goals may, and often do, make different decisions.

On the very first computers, the distinction between hardware and software was clear. The hardware carried out a small number of simple instructions, such as ADD and JUMP, and everything else was programmed explicitly. If a program needed to multiply two numbers, the programmer had to write his own multiplication procedure or use a library procedure. As time progressed, it became obvious to hardware designers that certain operations were being performed so frequently as to justify constructing special hardware circuits to execute them directly (to make them faster). The result was a trend toward moving operations downward, to a lower level. What had previously been programmed explicitly at the conventional machine level was later found below it in the hardware.

With the coming of age of microprogramming and multilevel computers, the reverse trend also became apparent. On the earliest computers there was no doubt that the ADD instruction was carried out directly by the hardware. On a microprogrammed computer, the conventional machine level's ADD instruction was interpreted by a microprogram running at the bottom level and was carried out as a series of small steps: fetch the instruction, determine its type, locate the data to be added,

fetch the data from memory, perform the addition, and store the result. This was an example of a function that moved upward, from the hardware level to the microprogram. Once again we emphasize: there are no hard and fast rules about what must be in hardware and what must be in software.

When developing a multilevel machine, the designers must decide what to put at each level. This is a generalization of the problem mentioned earlier, of deciding what to put in the hardware and what to put in the software, the hardware merely being the lowest level. It is interesting to note some of the features of some modern computers that are now performed by the hardware or microprogram but that originally were explicitly programmed at the conventional machine level. They include

1. Instructions for integer multiplication and division.
2. Floating-point arithmetic instructions (see Appendix B).
3. Double-precision arithmetic instructions (arithmetic on numbers with more significant figures than usual).
4. Instructions for calling and returning from procedures.
5. Instructions for speeding up looping.
6. Instructions for counting (adding 1 to a variable).
7. Instructions for handling character strings.
8. Features to speed up computations involving arrays (indexing and indirect addressing).
9. Features to permit programs to be moved in memory after they have started running (relocation facilities).
10. Clocks for timing programs.
11. Interrupt systems that signal the computer as soon as an input or output operation is completed.
12. The ability to stop one program and start another in a small number of instructions (process switching).

The point of this discussion is to show that the boundary between hardware and software is arbitrary and constantly changing. Today's software is tomorrow's hardware and vice versa. Furthermore, the boundaries between the various levels are also fluid. From the programmer's point of view, how an instruction is actually implemented is unimportant (except perhaps for its speed). A person programming at the conventional machine level can use its multiply instruction as though it were a hardware instruction, without having to worry about it or even be aware of whether it really is a hardware instruction or not. One man's hardware is another man's software.

The fact that a programmer need not be aware of how the level he is using is implemented leads to the idea of structured machine design. A level is often called a

virtual machine because the programmer thinks of it as a real physical machine, even though it does not actually exist; that is, it is "virtual." By structuring a machine as a series of levels, programmers working on level N need not be aware of all the messy details of the underlying levels. This structuring enormously simplifies producing complex (virtual) machines.

1.5. PROCESSES

A concept that is fundamental to understanding how computers are organized is the process concept. A **process** (sometimes called a **sequential process**) is basically a program in execution. It is an active entity, capable of causing events to happen. A process (i.e., a running program) may draw Chinese characters on the pen-and-ink plotter, or it may play chess with a human being sitting at a time-sharing terminal, or it may regulate production in an automated factory, among uncountably many other activities. A process is in contrast to a program, which is a passive entity. A program lying on someone's desk cannot do anything by itself.

One can draw an analogy between a process and a living animal. Both are capable of doing things. Both are capable of changing their environment in certain ways. On the other hand, both a program and a dead animal, or model of an animal, are passive and cannot cause events to happen. This discussion is certainly not meant to suggest that a process is alive but merely to point out that the difference between a process and a program has a certain similarity to the difference between a living and a nonliving animal.

At any instant of time, a process is in a certain state. The state of a process tells how far the process is in its computation. It contains all the information needed to stop the process and then restart it later. The process state consists of (at least) the following information:

1. The program.
2. An indication of which instruction is to be executed next.
3. The values of all the program's variables and data.
4. The status and position of all input/output devices being used.

As a process progresses in time, its state changes. Different instructions, in turn, become "the next instruction to be executed." The variables take on successive values. Although it is also possible for a process to modify its program during execution, doing so is considered poor programming practice. We will assume that the program is not changed by the process. It is often convenient to group all the changeable parts of a process' state together under the heading **state vector.** We can now regard a process as consisting of two parts: a program (which does not change) and a state vector (which does change). The program consists of instructions for changing the state vector.

As an example of a process moving from one state to another, consider the following PL/I program:

```
SIMPLE:  PROCEDURE OPTIONS(MAIN);
         DECLARE I FIXED INITIAL(0), J FIXED INITIAL(1);
L1:      I = I + 2;
L2:      J = I + J;
L3:      I = I * J + 4;
L4:      J = I * I;
END SIMPLE;
```

The state of this program during execution can be characterized by three items of information: I, J, and the instruction to be executed next. The state vector for a process executing this program is shown in Fig. 1-4.

A process passes through a time-ordered sequence of states. In order for a pro-

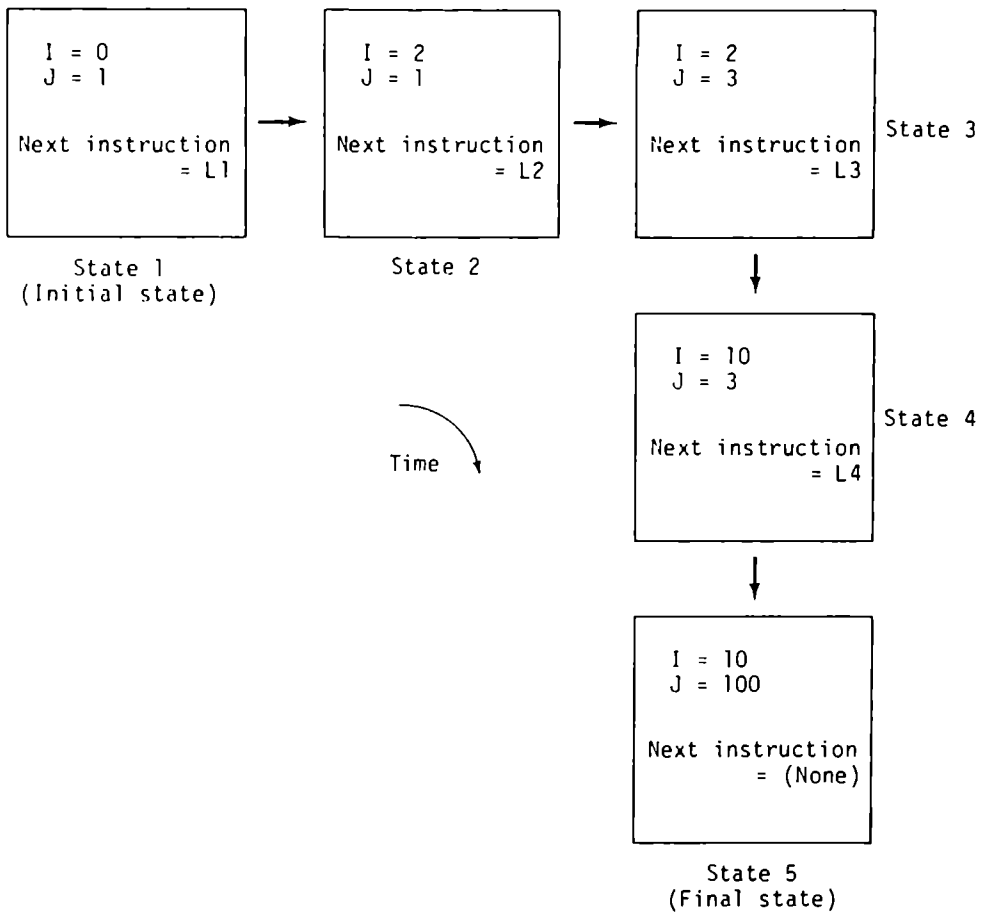


Fig. 1-4. The state vector for a process executing the procedure SIMPLE changes in time.

gram to be able to execute—that is, become a process—it must have a computer or **processor** to run on. The function of the processor is to advance the process from one state to the next. In other words, all a processor does is make changes in a process' state vector.

A process has two important properties:

1. The effect of a process is independent of its execution speed.
2. If a process executes again with the same data, it goes through precisely the same sequence of states and gives the same result.

These properties emphasize the sequential nature of a process.

The processor that moves a process from one state to the next state need not be a hardware machine. It could equally well be another process. When one process, P1, advances another process, P2, by changing P2's state vector, P1 is, in fact, interpreting P2's program. This means that P1 is a running interpreter.

The process concept provides another way of characterizing an interpreter. An interpreter is a program that, when executed, pushes the state vector of some process, P, through the sequence of states required by P's program. This activity amounts to executing P and is precisely the same thing a hardware processor would do.

Figure 1-5 shows a three-level computer on which level 3 is supported by an interpreter running on level 2, and on which level 2 is supported by an interpreter running at level 1. At each level are a program and a state vector. The hardware fetches instructions from P1 and changes SV1 appropriately after each instruction in order to execute P1. This level 1 process fetches the instructions of P2 and makes the corresponding changes to SV2 needed to execute P2. In other words, P1 interprets P2.

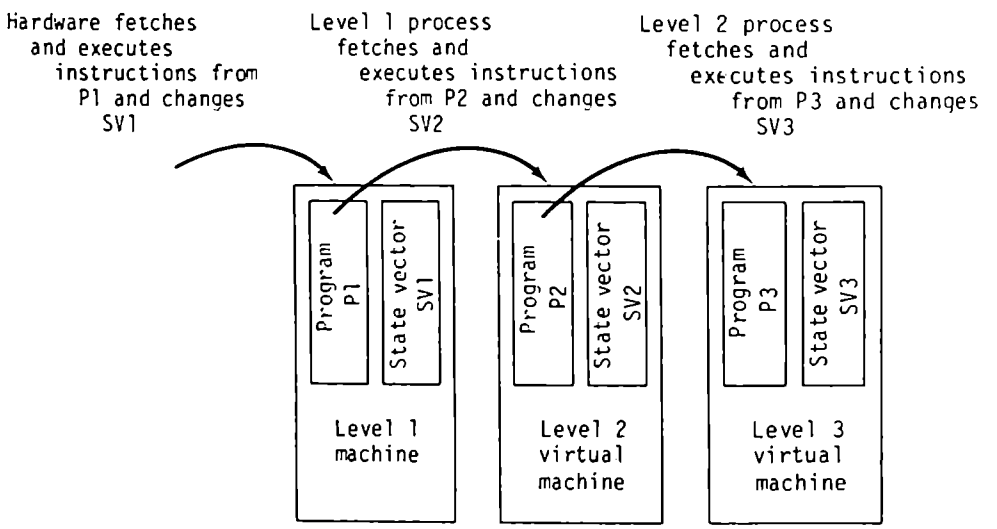

Hardware fetches and executes instructions from P1 and changes SV1
Level 1 process fetches and executes instructions from P2 and changes SV2
Level 2 process fetches and executes instructions from P3 and changes SV3
Program P1
State vector SV1
Program P2
State vector SV2
Program P3
State vector SV3
Level 1 machine
Level 2 virtual machine
Level 3 virtual machine

Fig. 1-5. A 3 level system with interpreters supporting levels 2 and 3.

Just as P1 interprets P2, P2 interprets P3. The program P3 might either be yet another interpreter or an applications program that solved some problem for one of the computer center's clients. This three-level system has three programs and three state vectors. At each level there is a "next instruction to be executed."

1.6. OUTLINE OF THIS BOOK

This book is about multilevel computers (which includes nearly all modern computers) and how they are organized. We will examine four levels in considerable detail—namely, the microprogramming level, the conventional machine level, the operating system machine level, and the assembly language level. Some of the basic issues that we will examine are as follows:

1. The overall design of the level (and why it was designed that way).
2. The kinds of instructions available.
3. The kinds of data used.
4. The mechanisms available for altering the flow of control.
5. The memory organization and addressing.
6. The relationship between instruction set and memory organization.
7. The method by which the virtual machine is implemented.

These topics are sometimes loosely grouped together under the imprecise labels of **computer organization** or **computer architecture.**

We are primarily concerned with concepts rather than details or formal mathematics. For that reason, some of the examples given will be highly simplified so as to emphasize the central ideas and not the details.

In order to provide some insight into how the ideas presented in this book can be, and are, applied in practice, we will examine three well-known computer families in some detail. These three families are the IBM 370 series, the CDC Cyber 70 series, and the DEC PDP-11 series. These three have been chosen for several reasons. First, all are widely used and the reader is likely to have access to at least one of them. Second, each one has its own unique architecture, which provides a basis for comparison and encourages a "what are the alternatives?" attitude. Books dealing with only one machine often leave the reader with a "true machine design revealed" feeling, which is absurd in light of the many compromises and arbitrary decisions that designers are forced to make. You are encouraged to study these and all other computers with a critical eye and to try to understand why things are the way they are, as well as how they could have been done differently, rather than simply accepting them as given.

It should be made clear from the beginning that this is not a book about how to program the IBM 370, Cyber 70, or PDP-11. These machines will be used for illu-

strative purposes where appropriate, but we make no pretense of being complete. Anyone wishing a thorough introduction to one of them should consult the manufacturer's publications.

Chapter 2 is an introduction to the basic components of a computer—processors, memories, and input/output equipment—and the techniques used to interconnect them and transmit information between them. This material has relevance to more than one level, and often to all levels of a multilevel system.

Chapters 3, 4, 5, and 6 each deal with one specific level shown in Fig. 1-2. Chapter 3 is about the conventional machine level. This makes a better starting place than the microprogramming level underlying it, partly because it is of more importance to the majority of programmers, partly because it is more familiar and better documented, and partly to better motivate the discussion of the microprogramming level that follows in Chapter 4.

Chapter 4 introduces the concepts of microprogramming and the architecture of the microprogramming level, plus their relation to the conventional machine that they support.

Chapter 5 covers some of the instructions, memory organization, and control mechanisms present at the operating system machine level.

Chapter 6 introduces the assembly language level. Because the assembly language level, unlike the levels below it, is usually implemented by translation rather than interpretation, the emphasis here is on the translation process instead of the details of any specific assembly languages. The related topics of macro expansion (which is itself a translation technique) and linking (which is really the last phase of the translation process) are also covered.

Chapter 7 takes a more global view of multilevel machines and investigates a variety of topics related to multilevel machines as a whole. Chapter 8 contains a list of suggested readings, arranged by subject, and an alphabetical list of literature citations. This is the most important chapter in the book. Use it.

The appendices contain brief introductions to finite-precision arithmetic and binary numbers, floating-point numbers, and boolean algebra.

PROBLEMS

1. Explain the following terms in your own words:
   a. machine language
   b. translator
   c. interpreter
   d. virtual machine
   e. process
   f. state vector
2. What is the difference between translation and interpretation?
3. Show all the states that a process executing the following program would have during its execution (assume that each line is a single, indivisible instruction).

```
PLAY: PROCEDURE OPTIONS(MAIN);
      DECLARE HAMLET FIXED INITIAL (100), MACBETH FIXED INITIAL (2);
S1:   HAMLET = 4 * MACBETH;
S2:   IF HAMLET < 6 THEN HAMLET = HAMLET + 6;
S3:   IF MACBETH > HAMLET THEN HAMLET = 2 * HAMLET;
S4:   MACBETH = HAMLET * HAMLET;
END PLAY;
```

4. Show all the states that a process executing the following program would have during its execution (assume that each line is a single, indivisible instruction).

```
DIET: PROCEDURE OPTIONS(MAIN);
      DECLARE WEIGHT FIXED, REASONABLE FIXED INITIAL (160);
S1:   WEIGHT = 200;
S2:   DO WHILE (WEIGHT > REASONABLE);
S3:   WEIGHT = WEIGHT - 10;
S4:   END;
END DIET;
```

5. Can the sequence of states through which a process passes be affected by the input data? Explain.

6. If the programs P1 and P2 of Fig. 1-5 are identical, what is the relationship between the machine languages of virtual machines 2 and 3?

7. Consider a four-level computer with identical interpreters at level 1, 2, and 3. It takes an interpreter N instructions to fetch, examine, and execute one instruction. A level 1 instruction takes K nanoseconds to be executed. How long does it take for an instruction to be executed at levels 2, 3, and 4?

8. Consider a four-level computer in which all the levels are different. Each level has instructions that are M times as powerful as those of the level below it; that is, one level r instruction can perform the same computation as M level $r - 1$ instructions. If a level 1 program requires K seconds to run, how long would equivalent programs take at levels 2, 3, and 4, assuming N level r instructions are required to interpret a single level $r + 1$ instruction?

9. Some instructions at the operating system machine level are identical to conventional machine language instructions. These instructions are carried out directly by the microprogram rather than by the operating system. In light of your answer to Problem 7, why do you think that such is the case?

10. In what sense are hardware and software equivalent? Not equivalent?

11. How many of the levels of Fig. 1-2 does your computer have? Which level does your reference manual describe?

12. If a computer has a card reader, what information about the card reader must be present in the state vector of a process running on it, in order to allow the process to be stopped at any arbitrary instant, saved, and then restarted on another identical computer?

2 COMPUTER SYSTEMS ORGANIZATION

A digital computer consists of an interconnected system of processors, memories, and input/output devices. This chapter is an introduction to these three components and to the methods by which they are interconnected. Although the components are, in fact, the computer's hardware, most of the topics discussed here are applicable to several other levels of a multilevel computer as well. The purpose of this chapter is to provide an overall view of how computers are organized, as background for the examination of specific levels in the four succeeding chapters.

2.1. PROCESSORS

The organization of a simple computer is shown in Fig. 2-1. The **central processing unit** (CPU) is the "brain" of the computer. Its function is to execute programs stored in the main memory by fetching their instructions, examining them, and then executing them one after another. The CPU is composed of several distinct parts. The control unit is responsible for fetching instructions from main memory and determining their type. The arithmetic and logical unit performs operations, such as addition and boolean AND, needed to carry out the instructions.

The CPU also contains a small, high-speed memory used to store temporary results and certain control information. This memory consists of a number of **registers,** each of which has a certain function. The most important register is the **program counter** (PC), which points to the next instruction to be executed. There is also an **instruction register** (IR), which holds the instruction currently being executed. Most

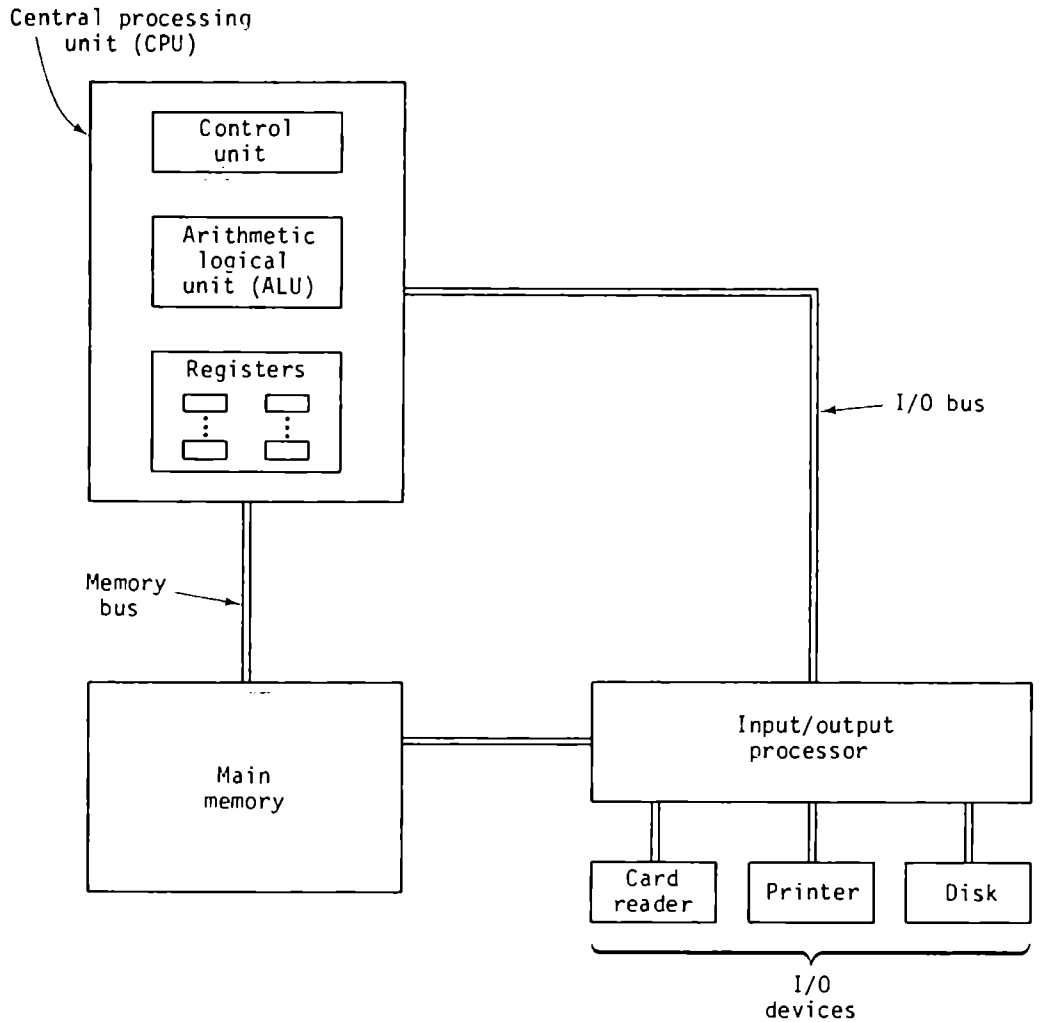


Fig. 2-1. The organization of a simple computer with one CPU and one i/o processor.

computers have other registers as well, some of which are available to the level 2 and level 3 programmers for storing intermediate results.

2.1.1. Instruction Execution

The CPU executes each instruction in a series of small steps:

1. Fetch the next instruction from memory into the instruction register.
2. Change the program counter so it points to the following instruction.
3. Determine the type of instruction just fetched.

4. See if the instruction requires any data from memory, and if so, determine where in memory the data is located.
5. If the instruction needs data from memory, load it into internal CPU registers.
6. Execute the instruction.
7. Store the results in the proper place.
8. Go to step 1 to begin executing the following instruction.

The description of how a CPU works closely resembles a program written in English. Figure 2-2 shows this informal program rewritten as a PL/I procedure. The

```
INTERPRETER:   PROCEDURE(MEMORY,AC,STARTING_ADDRESS);

/* THIS PROCEDURE INTERPRETS PROGRAMS FOR A SIMPLE MACHINE WITH 1
   INSTRUCTION PER WORD.  THE MEMORY CONSISTS OF A SEQUENCE OF
   WORDS NUMBERED 0,1,...4095. THE MACHINE HAS A PROCESSOR
   REGISTER CALLED THE AC, USED FOR ARITHMETIC. THE ADD
   INSTRUCTION ADDS A MEMORY WORD TO THE AC, FOR EXAMPLE. THE
   INTERPRETER KEEPS RUNNING UNTIL THE RUN BIT IS TURNED OFF (BY
   A HALT INSTRUCTION). THE STATE OF A PROCESS RUNNING ON THIS
   MACHINE CONSISTS OF: MEMORY, THE PROGRAM COUNTER, THE RUN BIT,
   AND THE AC. THE INITIAL STATE IS PASSED VIA THE PARAMETERS. */

DECLARE (MEMORY(0:4095),AC,STARTING_ADDRESS) FIXED BINARY;
DECLARE (PROGRAM_COUNTER, INSTRUCTION_REGISTER,RUN_BIT,TYPE,DATA,
        DATA_LOCATION) FIXED BINARY;

PROGRAM_COUNTER = STARTING_ADDRESS;

RUN_BIT = 1;

DO WHILE (RUN_BIT = 1);
   /* FETCH NEXT INSTRUCTION INTO THE INSTRUCTION REGISTER */
   INSTRUCTION_REGISTER = MEMORY(PROGRAM_COUNTER);

   /* ADVANCE THE PROGRAM COUNTER TO POINT TO THE NEXT INSTR */
   PROGRAM_COUNTER = PROGRAM_COUNTER + 1;

   /* DECODE INSTRUCTION AND SET TYPE TO APPROPRIATE VALUE */
   CALL DETERMINE_INSTRUCTION_TYPE(INSTRUCTION_REGISTER,TYPE);

   /* LOCATE DATA USED BY INSTRUCTION. -1 MEANS NO DATA. */
   CALL FIND_DATA(INSTRUCTION_REGISTER,DATA_LOCATION);

   /* FETCH DATA FROM MEMORY IF NEEDED */
   IF DATA_LOCATION >= 0 THEN DATA = MEMORY(DATA_LOCATION);

   /* ADVANCE THE PROCESS TO ITS NEXT STATE */
   CALL EXECUTE(TYPE,DATA,MEMORY,AC,PROGRAM_COUNTER,RUN_BIT);
END;
RETURN;

END INTERPRETER;
```

Fig. 2-2. An interpreter for a simple computer.

very fact that it is possible to write a program that can imitate the function of a CPU shows that a program need not be executed by a "hardware" CPU consisting of a box full of electronics but that a program can also be carried out by having another program fetch, examine, and execute its instructions. A program (such as Fig. 2-2) that fetches, examines, and executes the instructions of another program is called an interpreter, as mentioned in Chapter 1.

This equivalence between hardware processors and interpreters has important implications for computer organization. After having specified the machine language, L, for a new computer, the design group can decide whether they want to build a hardware processor to execute programs in L directly or whether they want to write an interpreter instead. If they choose to write an interpreter, they must also provide some machine to run the interpreter.

Because an interpreter breaks the instructions being executed into small steps, the machine on which the interpreter runs can often be much simpler, and therefore less expensive, than hardware processor for the interpreted machine would be. For economic as well as other reasons, programs at the conventional machine level of most modern computers are carried out by an interpreter running on a totally different and relatively simple level 1 machine, which we shall call the microprogramming level.

The collection of all instructions available to the programmer at a level is called the **instruction set** of that level. The number of instructions in the instruction set varies from machine to machine and level to level. For the conventional machine level, for example, the size of the instruction set is typically in the range 20 to 300. A large instruction set is not necessarily better than a small one. In fact, the opposite tends to be true. A large instruction set often indicates that the instructions are not very general. Compilers for high-level languages, such as ALGOL 68, COBOL, FORTRAN, PASCAL, and PL/I, generally perform better on machines with small, well-chosen instruction sets than on machines with large, unwieldy ones.

It is important to realize that instruction set and organization of the microprogramming level is, in fact, the instruction set and organization of the hardware, that is, the CPU. The instruction set and organization of the conventional machine level, in contrast, is determined by the microprogram, and not by the hardware.

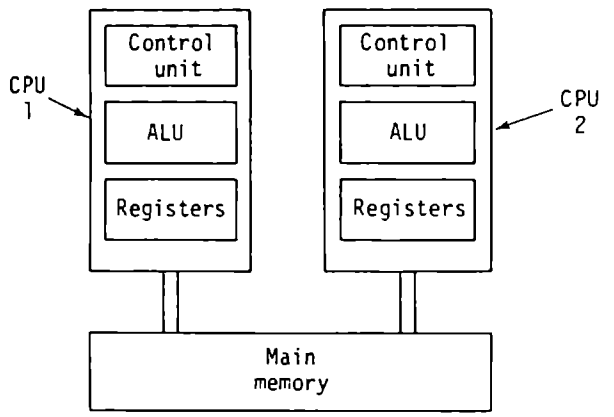
Not all processors are general-purpose CPUs. Computer systems often contain one or more processors with a specific function that requires a limited, specially designed instruction set. Such special-purpose processors are widely used to perform input/output. As an example, consider a graphics terminal, a device with a screen similar to a television set's, on which the computer can display drawings of electrical circuits, blueprints of buildings, maps, graphs of data, and many other kinds of useful information.

A graphics terminal normally contains a highly specialized processor and a memory. The graphics processor may have instructions to plot points in several intensities and colors; draw solid, dotted, and dashed lines; display characters; and produce geometric figures, such as squares, circles, and triangles. Such an instruction set is obviously very different from one a general-purpose CPU needs.

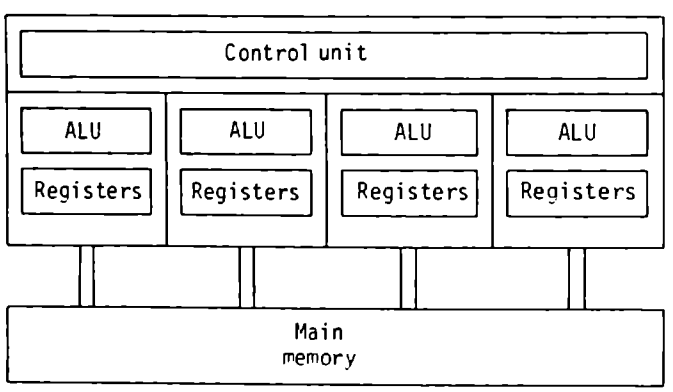
2.1.2. Parallel Instruction Execution

Although early computers had a single CPU that executed instructions one at a time, certain more advanced designs increase the effective computing speed by allowing several instructions to be executed at the same time. The simplest method is to have two or more independent CPUs sharing a common memory, as shown in Fig. 2-3(a). Usually each CPU has its own program to run, with no interaction between the CPUs. A computer with multiple processors is called a **multiprocessor** system.

Some problems require carrying out the same computation on many sets of data. For example, a weather prediction program might read hourly temperature measurements taken from 1000 weather stations and then compute the daily average at each

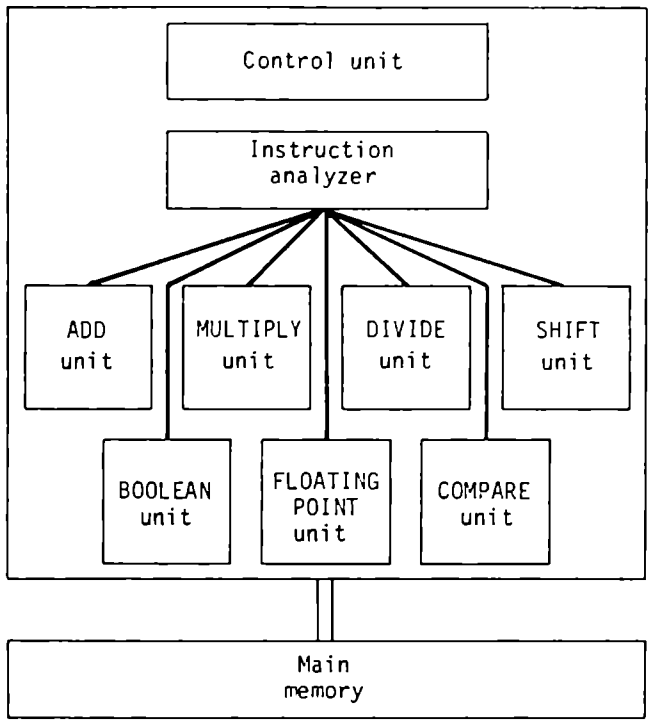


(a)

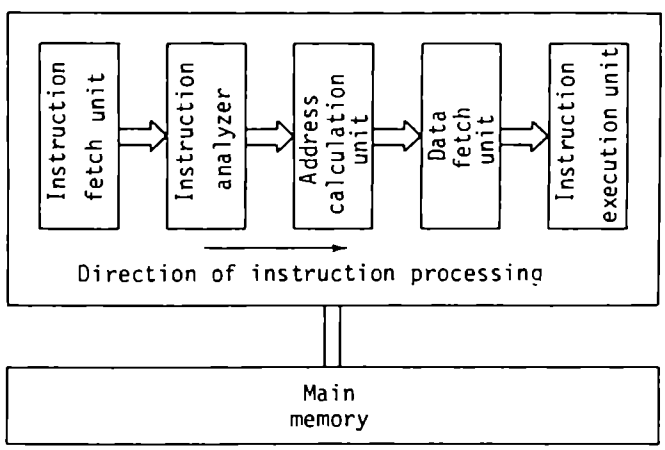


(b)

Fig. 2-3. Four different kinds of parallel machines. (a) Multiple independent CPU's. (b) An array processor.



(c)



(d)

Fig. 2.3. (cont.) (c) A single CPU with multiple functional units. (d) A pipeline machine.

station by performing exactly the same computation on each set of 24-hourly readings. For each station's data, it would load the first value into a register, then add the second value, then the third, and so on. Finally, it would divide the sum by 24. Since the same program is used on each data set, a processor with one program counter and one instruction decoder but N arithmetic units and N register sets could carry out computations on N data sets simultaneously.

This configuration, which is shown in Fig. 2-3(b), is sometimes called a single-instruction-stream, multiple-data-stream processor, or an **array processor.** An example of this type of organization is the ILLIAC IV, designed at the University of Illinois and constructed by the Burroughs Corporation. The ILLIAC IV consists of four control units, each of which operates on 64 data sets simultaneously. The ILLIAC IV can perform four different computations at the same time, each computation being carried out on 64 sets of data, making a total of 256 calculations in parallel.

The CDC 6600 and Cyber 74 use yet another form of parallel instruction execution. These machines have separate arithmetic units for addition, multiplication, division, and other operations, for a total of 10 separate units (including two of some units). There is a single control unit that fetches and decodes instructions. As soon as the type of an instruction is known, it is sent to the appropriate unit (e.g., multiplication), and the fetching and decoding of the next instruction begin, concurrently with the execution of the first instruction. A well-written program may have up to 10 instructions being executed simultaneously. This system only makes sense if the time needed to fetch and decode an instruction is very small compared to the time needed to carry it out, which is the case on the CDC machines. Figure 2-3(c) illustrates this concept.

A related processor organization is the **pipeline** machine, shown in Fig. 2-3(d). In this design, each of the steps in executing an instruction (e.g., fetch instruction into the instruction register, determine instruction type, locate the data, etc.) has a separate unit within the CPU to perform it. When the machine starts, the initial instruction is fetched by the first unit. Then the second unit begins decoding the initial instruction while the first unit is busy fetching the second instruction. A little later, the third unit is busy examining the initial instruction to see if it needs data from memory, while at the same time the second unit is decoding the second instruction and the first unit is busy fetching the next instruction. The IBM 360/195, CDC STAR, and the University of Manchester MU5 use sophisticated forms of pipelining.

A pipeline machine can be compared to an automobile assembly line: at each position along the (dis)assembly line, the same function is performed on the stream of instructions that come by. The ones near the end of the pipeline are nearly completed; the ones near the beginning of the pipeline are barely started.

2.2. MEMORY

The **memory** is that part of the computer where programs and data are stored. Some computer scientists use the term **store** or **storage** rather than memory. Without

a memory from which the processors can read and write information, there would be no stored program digital computers as we know them.

2.2.1. Bits

The basic unit of memory is the binary digit, called a **bit.** A bit may contain a 0 or a 1. This is the simplest possible unit. (A device capable of storing only zeros could hardly form the basis of a memory system.)

People often say that computers use binary arithmetic because it is "efficient." What they mean (although they rarely realize it) is that digital information can be stored by distinguishing between different values of some continuous physical quantity, such as voltage or current. The more values that must be distinguished, the smaller the separation between adjacent values, and the less reliable the memory. The binary number system requires only two values to be distinguished. Consequently, it is the most reliable system for encoding digital information. If you are unfamiliar with binary numbers, see Appendix A.

Some computers are advertised as having decimal rather than binary arithmetic. This can be accomplished by using four bits to store one decimal digit. Four bits provide 16 combinations, used for the 10 digits 0 through 9, with 6 combinations not used. The number 1944 is shown below encoded in decimal and in pure binary, using 16 bits in each example:

Decimal	Binary
0001 1001 0100 0100	0000011110011000

Sixteen bits in the decimal format can store the numbers from 0 to 9999, giving only 10,000 combinations, whereas a 16-bit pure binary number can store 65,536 different combinations. This is why people say that binary is more efficient.

However, consider what would happen if some brilliant young electrical engineer invented a highly reliable electronic device that could directly store the digits 0 to 9 by dividing the region from 0 to 10 volts into 10 parts. Four of these devices could store any decimal number from 0 to 9999. Four such devices would provide 10,000 combinations. They could also be used to store binary numbers, by only using 0 and 1, in which case, 4 of them could only store 16 combinations. With such devices, the decimal system is obviously more efficient.

2.2.2. Memory Addresses

Memories consist of a number of **cells** (or **locations**), each of which can store a piece of information. Each cell has a number, called its **address**, by which programs can refer to it. If a memory has N cells, they will have addresses 0 to N − 1. All cells in a memory contain the same number of bits. If a cell consists of K bits, it can hold any one of 2^K different bit combinations. Figure 2-4 shows three different organiza-

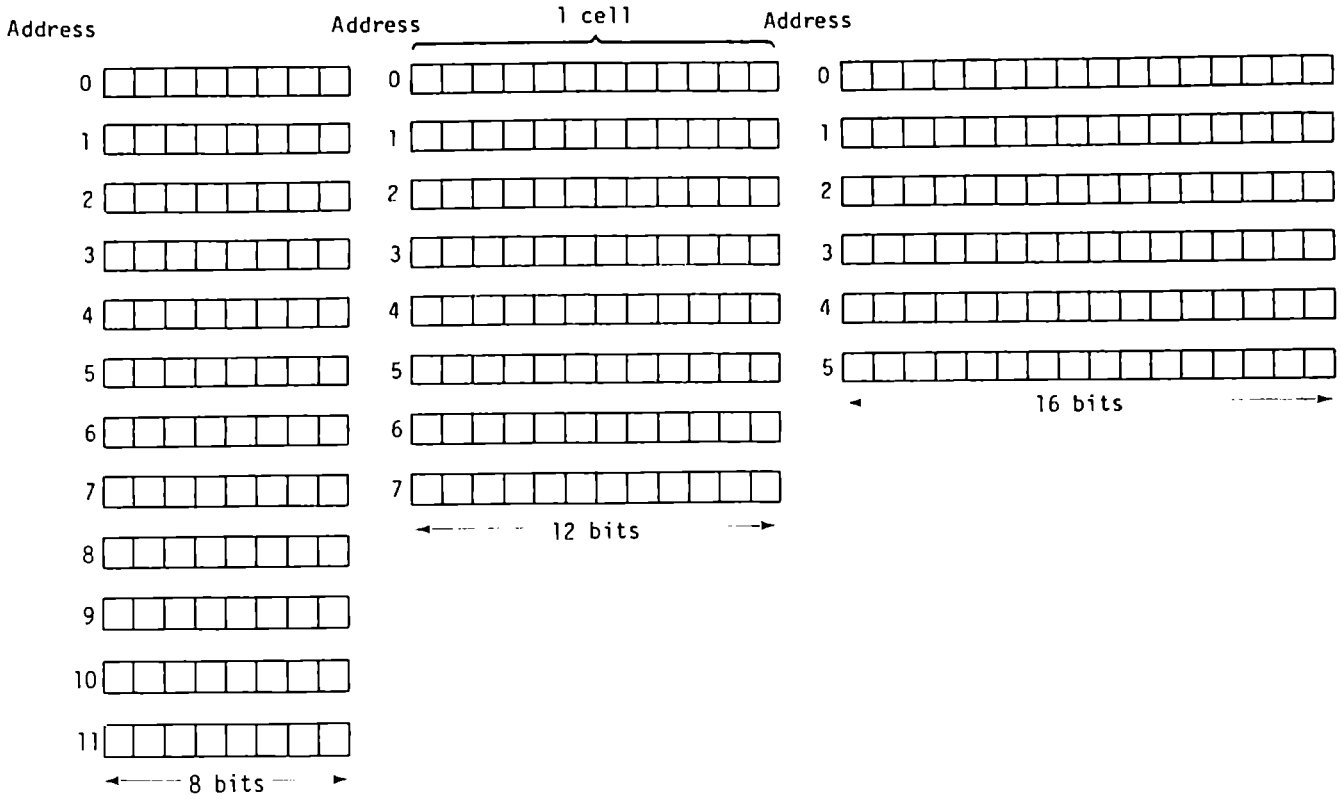


Fig. 2-4. Three different ways of organizing a 96-bit memory into cells.

tions for a 96-bit memory. Note that adjacent cells have consecutive addresses (by definition).

Computers that use the binary number system (including octal and hexadecimal notation for binary numbers) also express memory addresses as binary numbers. If an address has M bits, the maximum number of cells directly addressable is 2^M. For example, an address used to reference the memory of Fig. 2-4(a) would need at least four bits in order to express all the numbers from 0 to 11. A three-bit address would be sufficient for Fig. 2-4(b) and Fig. 2-4(c), however. The number of bits in the address is related to the maximum number of directly addressable cells in the memory and is independent of the number of bits per cell. A memory with 2^{12} cells of eight bits each and a memory with 2^{12} cells of 60 bits each would each need 12-bit addresses.

The number of bits per cell for some computers that have been sold commercially follows.

Burroughs B1700	1 bit per cell
IBM 370	8 bits per cell
DEC PDP-8	12 bits per cell
IBM 1130	16 bits per cell
DEC PDP-15	18 bits per cell
XDS 940	24 bits per cell
Electrologica X8	27 bits per cell
XDS SIGMA 9	32 bits per cell
Honeywell 6180	36 bits per cell
CDC 3600	48 bits per cell
CDC Cyber 70	60 bits per cell

On some machines, the term **word** is used in place of cell. On other machines, especially machines with eight bits per cell, the term **byte** is used instead of cell, and the term "word" is reserved for two bytes, or four bytes, depending on the machine. This situation, unfortunately, can lead to much confusion.

The significance of a cell is that it is the smallest addressable unit. Each cell has a unique address, and consecutive cells have addresses differing by 1. The significance of a word is that, at level 2, registers that hold one word are usually available for temporary storage. Furthermore, there are instructions to fetch and store an entire word in one operation. Following popular usage, we will use the term "word" rather than "cell" to indicate the basic information unit of memory, except where the distinction is significant. You should keep in mind that the relation between words and cells depends on the machine in question.

The CPU communicates with the main memory by means of two registers—the **memory address register** (MAR) and the **memory buffer register** (MBR). When the CPU needs to read a word from memory, either an instruction or data, it loads the address of the word it wants into the MAR and sends a read signal to the memory. The memory begins operating; and after one memory cycle, it puts the desired word in the MBR, where the CPU can access it. To write a word into memory, the CPU puts the address to be stored into, in the MAR and the word to be stored in the

MBR. Then it signals the memory to begin a store operation. Except on a few computers, the MAR and MBR are not directly accessible to level 2 programs, although they are always accessible to level 1 programs.

2.2.3. Metabits

Although main memory is used to store both programs and data, the processor generally has no way of telling whether a particular cell contains an instruction or data. A common programming error is for a program to jump into the data accidently and try to execute it. Inadvertent attempts to execute data (or to multiply one machine instruction by another) could be caught immediately by the processor if each cell (or word) had an extra bit associated which it, containing a 0 for instructions and a 1 for data, as shown in Fig. 2-5.

An extra bit associated with each cell (or word) used to indicate its contents is called a **metabit**. The following method shows how the metabits of Fig. 2-5 could be used to detect some programming errors. When the processor fetched a cell (or word) from memory, either data or instruction, the metabit associated with that memory word would be loaded into a special metabit register within the processor. If the

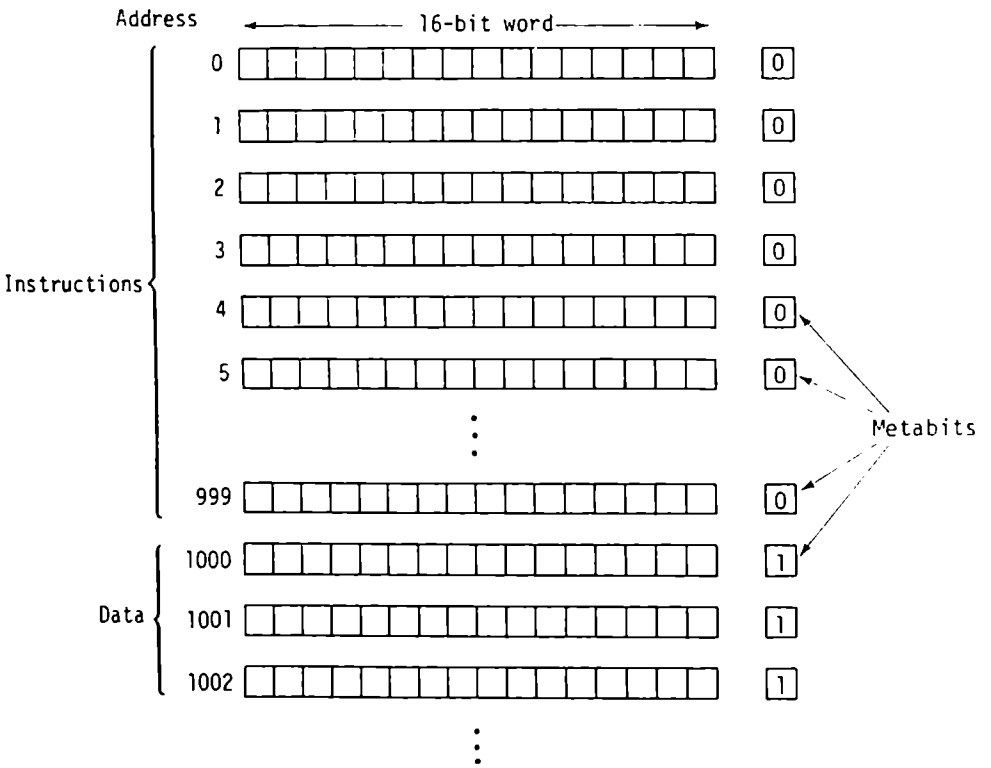


Fig. 2-5. A memory consisting of words with 16 data bits and one metabit. The metabit is 0 for instructions and 1 for data.

metabit register contained a 1 after an instruction fetch or a 0 after a data fetch, the processor would stop execution and print an error message.

Metabits can be used for other purposes as well. For example, if each cell (or word) had several metabits instead of just one, the metabits could not only distinguish between instructions and data but, for the latter, could also specify the type of data. Characters, integers, floating-point numbers (see Appendix B), decimal numbers, and other data types could each be uniquely indicated by the metabits. Doing so would eliminate the need for different instructions for integer addition, floating-point addition, and decimal addition. There could be a single addition instruction, and the processor would examine the metabits to see which kind was needed. A further extension would provide metabits (a) to indicate integer arrays, floating-point arrays, and decimal arrays as well as other data types, and (b) to allow the addition instruction to operate on whole arrays as well as individual numbers.

2.2.4. Secondary Memory

Because every word in main memory is directly accessible in a very short time, main memory is expensive. Consequently, most computers have slower, cheaper, and usually much larger secondary memories as well. Secondary memories are often used to hold very large sets of data.

Magnetic Tapes

Historically, **magnetic tape** was the first kind of secondary memory. A computer tape drive is analogous to a home tape recorder: a 2400-ft-long tape is wound from the feed reel past a recording head to the take-up reel. By varying the current in the recording head, the computer can write information on the tape in the form of little magnetized spots.

Figure 2-6 shows how information is organized on a magnetic tape. On a computer with eight-bit bytes, each frame contains one byte, plus an extra, redundant

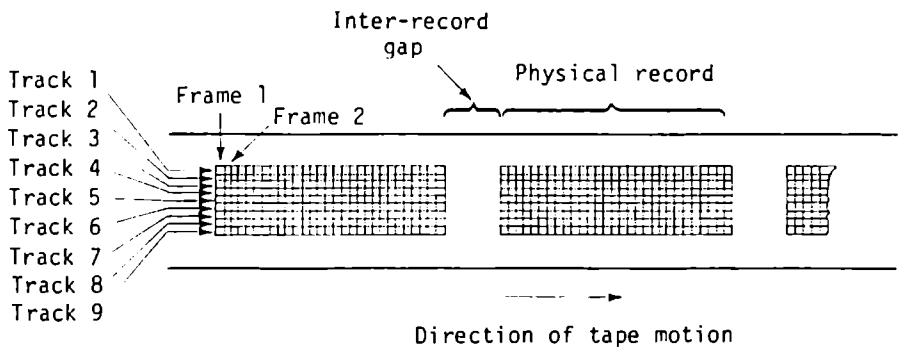


Fig. 2-6. Information on a magnetic tape is recorded as a sequence of rectangular bit matrices.

bit, called a parity bit, to improve reliability. Typical recording density is 1600 frames per inch, which means that the distance between frames is less than 1/1000 of an inch. After a tape drive has finished writing a record, it leaves a gap on the tape while slowing down. If the program writes very short physical records on the tape, most of the space will be wasted in the gaps. By writing physical records that are much longer than the gap, the tape utilization can be kept high.

Magnetic tapes are sequential-access devices. If the tape is positioned at the beginning, to read physical record N, it is first necessary to read physical records 1 through N − 1, one at the time. If the information desired is near the end of the tape, the program will have to read almost the entire tape, which may take several minutes. Forcing a CPU that can execute millions of instructions per second to wait 200 sec while a tape is advanced is wasteful. Tapes are most appropriate when the data must be accessed sequentially.

Magnetic Disks

A **disk** is a piece of metal, about the same size and shape as an LP phonograph record, to which a special magnetizable coating has been applied at the factory, generally on both sides. Information is recorded on a number of concentric circles, called **tracks** (see Fig. 2-7). Disks typically have a few hundred tracks per surface. There is a movable head that can be moved closer to, or farther from, the center. The head is wide enough to read or write information from exactly one track. A disk drive often has several disks stacked vertically about an inch apart. In such a configuration the arm will have one head next to each surface, all of which move in and out together. The radial position of the heads (distance from the spindle) is called the **cylinder address**. A disk drive with N disks will have 2N surfaces, hence 2N tracks per cylinder.

Tracks are divided into **sectors**, normally between 10 and 100 sectors per track. A sector consists of a certain number of machine words, typically 32 to 256. On some disks, the number of sectors per track can be set by the program.

Each disk drive has a small special-purpose computer associated with it, called the disk **controller**. The controller helps to transfer information between main memory and the disk. In specifying a transfer, the program must provide the cylinder and surface, which together specify a unique track, the sector number where the information starts (disk transfers always begin at the first word of a sector, never in the middle), the number of words to be transmitted, the main memory address where the information comes from or goes to, and whether information is to be read from the disk or written onto it.

If the head happens to be positioned over the wrong cylinder, it must first be moved. This step is called a **seek**. A seek typically takes 5 msec between adjacent tracks and 50 msec to go from the innermost cylinder to the outermost cylinder. Once the head is positioned properly, the controller must wait until the first sector has rotated under the head before beginning the transfer. The time wasted waiting

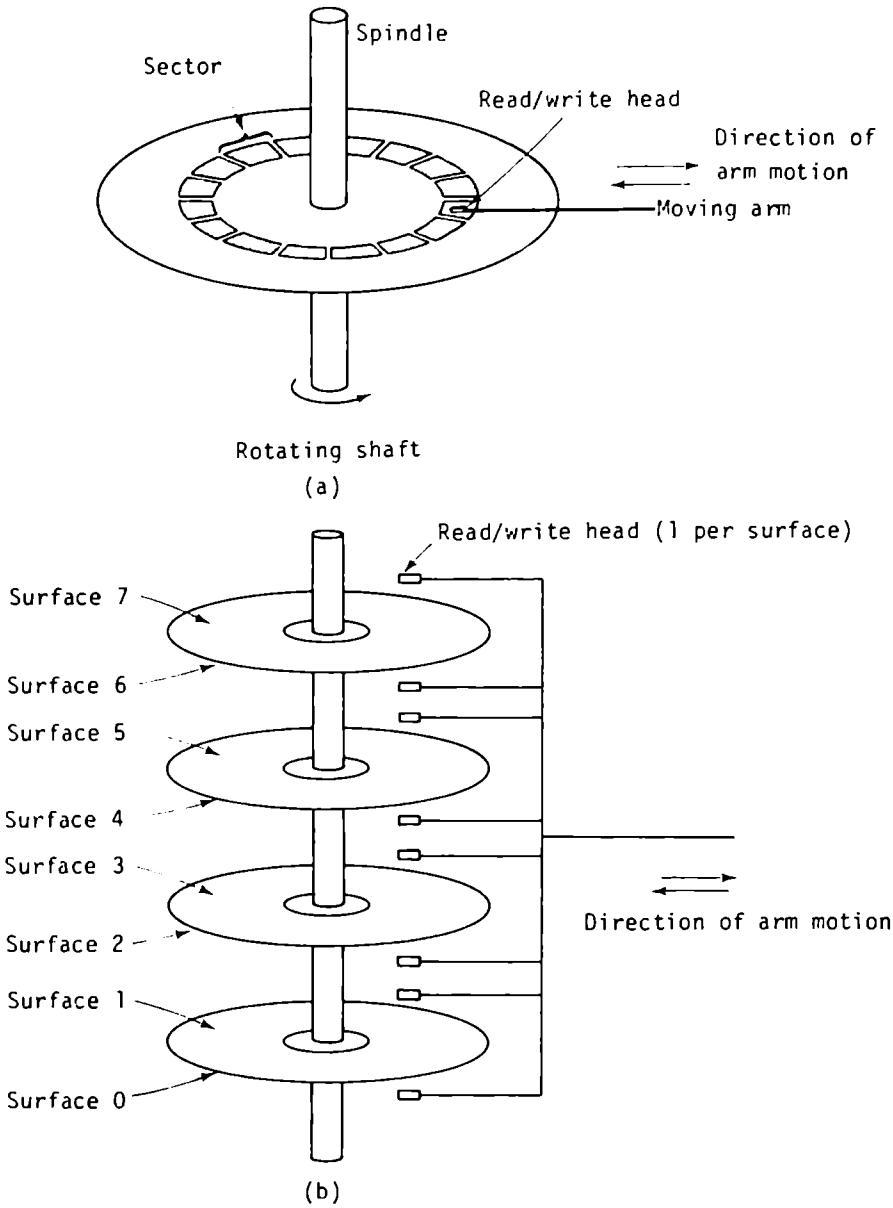


Fig. 2-7. A disk. (a) A view from above. (b) Side view.

for the right sector varies from 0, if the program is very lucky, to the complete rotation time if it just missed. The total access time is the seek time plus the rotational waiting time. The information is transferred at a rate of one track per rotation period. Some disk drives are constructed so that the disks themselves can be removed and brought to another computer, or stored, just like a tape.

Magnetic Drums

A variation on the disk is the **drum**, a cylinder on which information can be recorded magnetically (see Fig. 2-8). Along the length of the drum are many fixed read/write heads. Each head can read or write one track. The tracks are again divided into sectors. Since the heads do not move, there is no seek time. Furthermore, several heads may be reading or writing in parallel. Drums have a smaller capacity than disks, but access is much faster because there is no seek time. Some drums have two or more sets of heads, spaced uniformly around the circumference of the drum. With two sets of heads, a given sector will always appear under one set of heads or the other within, at most, one-half of the rotation period. Fixed-head disks are logically similar to drums in that the heads do not move but have the physical appearance of a disk.

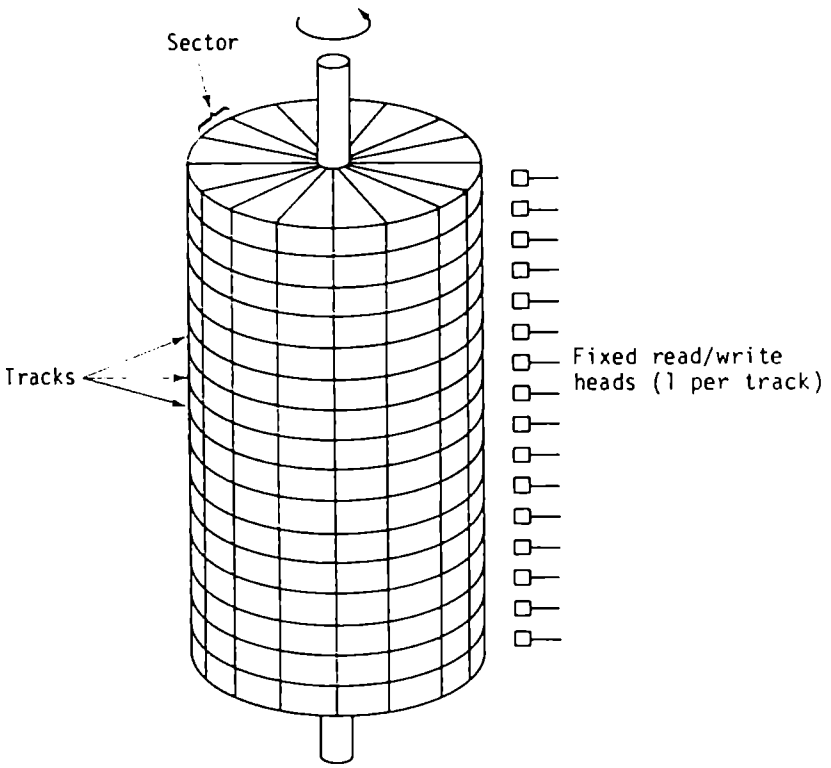


Fig. 2-8. A drum.

Optical Memories

In recent years there has been a great deal of interest in optical as opposed to magnetic memory. Optical memories can have very high recording densities (a strip of ordinary 35-mm black-and-white film 3 ft long can hold more information than a

magnetic tape 2400 ft long). Information recorded photographically is permanent and cannot be rewritten, as magnetic memories can. Some optical memories use lasers to achieve high reliability.

2.3. INPUT/OUTPUT

Before a computer can get to work solving a problem, it must be given the program, plus the data if there is any. After it has found the solution, the computer must communicate this solution to the human beings who posed the problem in the first place. The topic of getting information into and out of computers is called input/output, or usually just i/o.

Not all input comes from people and not all output is intended for people. A computer-operated solar telescope may get its input data directly from instruments observing the Sun. A computer controlling an automated chemical plant may direct its output to machines throughout the plant that regulate the chemicals being produced.

2.3.1. I/O Devices

Literally hundreds of kinds of i/o devices are available today, and the number is growing rapidly. A few of the more common ones are listed below.

Modified electric typewriters (teletypewriters)

Paper tape readers

Paper tape punches

Card readers

Card punches

High-speed line printers

Cathode-Ray Tube (CRT) displays

Optical readers that read printed matter, such as books

Mark sensors that can read cards marked with a special pencil

Experimental equipment, such as cyclotrons or telescopes

Pen and ink plotters that draw graphs

Magnetic ink readers that read bank checks

Rats and other laboratory animals being used for psychology experiments

2.3.2. I/O Processors

Some i/o devices can transmit a large amount of data in a short time. If the CPU had to process every character separately, a great deal of CPU time would be wasted.

To avoid tying up the CPU for long periods of time on i/o, most medium and large computers have one or more specialized, low-cost i/o processors. Because the i/o is performed by these special processors, the CPU is available to spend most of its time on more difficult computations. The i/o processors can run in parallel with the CPU. In other words, while the CPU is busy computing, the i/o processors can be doing i/o. IBM calls its i/o processors **channels**. CDC calls its i/o processors **peripheral processing units**, or **PPUs**.

2.3.3. Character Codes

Each computer has a set of characters that it can input and output. As a bare minimum, this set includes the 26 capital letters, the 10 digits 0 through 9, and a few punctuation marks, such as space, period, minus sign, comma, and carriage return. More sophisticated character sets include both capital letters and small letters, the 10 digits, a wide assortment of punctuation marks, a collection of special characters useful in mathematics and business, and sometimes even Greek letters.

In order to transfer these characters into the computer, each one is assigned a number, for example, a = 1, b = 2, . . . , z = 26, + = 27, − = 28, and so on. The mapping of characters onto integers is called a **character code**. Present-day computers use either a 6, 7, 8, or 9-bit code. A six-bit code allows only $2^6 = 64$ characters – namely, 26 letters, 10 digits, and 28 other characters, mostly punctuation marks and mathematical symbols.

For many applications, 64 characters are not enough; then a seven- or eight-bit character code must be used. A seven-bit character code allows up to 128 characters. One widely used seven-bit code is called ASCII (American Standard Code for Information Interchange). Figure 2-9 shows the ASCII code. Codes 1 to 37 (octal) are control characters and do not print. The most widely used eight-bit character code is the IBM EBCDIC code.

2.3.4. Error-Correcting Codes

Computers, like human beings, are inclined to make errors occasionally. In particular, CPUs and main memory do not have any moving parts and thus are highly reliable; but i/o devices usually involve physical motion and so are less reliable. Errors can be caused by specks of dust on the reading heads of a paper tape reader, card reader, magnetic tape drive, or disk. They can also be caused by attempting to read old, dirty, or warped cards, not to mention cards that have been bent, folded, spindled, or mutilated (unintentionally or otherwise). Data that is transmitted over a telephone line may be received incorrectly if the line is noisy. Electrical-power surges can cause errors. In short, when characters or other data must be input, output, or transmitted from one place to another, errors can occur. In this section we will investigate character codes that allow errors to be detected and, in some cases, even corrected.

A very simple, but widely used, method for detecting single errors—that is, one bit changed from a 0 to a 1 or a 1 to a 0—is to add a **parity** bit to each character.

Control characters

0	NUL	Null	20	DLE	Data link escape
1	SOH	Start of heading	21	DC1	Device control 1
2	STX	Start of text	22	DC2	Device control 2
3	ETX	End of text	23	DC3	Device control 3
4	EOT	End of transmission	24	DC4	Device control 4
5	ENQ	Enquiry	25	NAK	Negative acknowledge
6	ACK	Acknowledge	26	SYN	Synchronous idle
7	BEL	Bell	27	ETB	End of transmission block
10	BS	Backspace	30	CAN	Cancel
11	HT	Horizontal tab	31	EM	End of medium
12	LF	Line feed	32	SUB	Substitute
13	VT	Vertical tab	33	ESC	Escape
14	FF	Form feed	34	FS	File separator
15	CR	Carriage return	35	GS	Group separator
16	SO	Shift out	36	RS	Record separator
17	SI	Shift in	37	US	Unit separator

40	(Space)	60	0	100	@	120	P	140	`	160	p
41	!	61	1	101	A	121	Q	141	a	161	q
42	"	62	2	102	B	122	R	142	b	162	r
43	#	63	3	103	C	123	S	143	c	163	s
44	$	64	4	104	D	124	T	144	d	164	t
45	%	65	5	105	E	125	U	145	e	165	u
46	&	66	6	106	F	126	V	146	f	166	v
47	'	67	7	107	G	127	W	147	g	167	w
50	(	70	8	110	H	130	X	150	h	170	x
51	)	71	9	111	I	131	Y	151	i	171	y
52	*	72	:	112	J	132	Z	152	j	172	z
53	+	73	;	113	K	133	[	153	k	173	{
54	,	74	<	114	L	134	\	154	l	174	|
55	-	75	=	115	M	135	]	155	m	175	}
56	.	76	>	116	N	136	^	156	n	176	~
57	/	77	?	117	O	137	_	157	o	177(Delete)	

Fig. 2-9. The ASCII character code (octal).

In an odd-parity code, the parity bit is chosen so that the number of 1 bits in the character, including the parity bit, is an odd number. In an even-parity code, the parity bit is chosen so that the number of 1's in the character, including the parity bit, is an even number. Figure 2-10 illustrates the creation of an eight-bit even-parity code from seven-bit ASCII characters. If one bit is changed during transmission, the number of 1's in the received character will have the wrong parity, and the receiver will know an error has occurred.

A method not only for detecting errors but for correcting them as well was devised by Richard Hamming (1950) and is known as a **Hamming code**. In a Hamming code, K parity bits are added to an N-bit character, forming a new character of length N + K bits. The bits are numbered starting at 1, not 0, with bit 1 the leftmost bit.

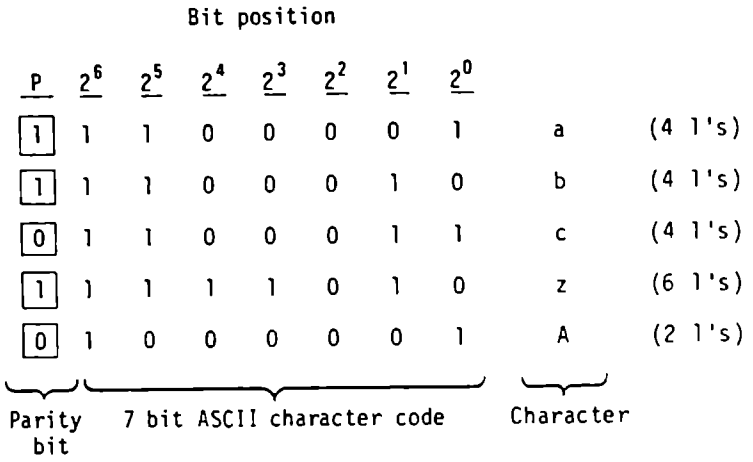


Fig. 2-10. A parity bit can be added to a 7-bit ASCII character to insure that every character contains an even number of 1's.

All bits whose bit number is a power of 2 are parity bits, and all the remaining bits are used for data. For a seven-bit ASCII character, four parity bits are added. Bits 1, 2, 4, and 8 are parity bits, and bits 3, 5, 6, 7, 9, 10, and 11 are the seven data bits. A Hamming code can be used on characters or messages of any length, although we will illustrate its use with seven-bit ASCII characters (and even parity).

Each parity bit checks specific bit positions, and the parity bit is set so that the total number of 1's in the checked positions is even. The bit positions checked by the parity bits are

bit 1 checks bits 1, 3, 5, 7, 9, 11

bit 2 checks bits 2, 3, 6, 7, 10, 11

bit 4 checks bits 4, 5, 6, 7

bit 8 checks bits 8, 9, 10, 11

In general, bit N is checked by those bits $b_1, b_2, \ldots, b_k$ such that $b_1 + b_2 + \cdots + b_k = N$. For example, bit 5 is checked by bits 1 and 4 because $1 + 4 = 5$. Bit 6 is checked by bits 2 and 4 because $2 + 4 = 6$. Figure 2-11 shows construction of a Hamming code for the character "b."

The ASCII character "b" is represented by the binary number 1100010. The Hamming code for "b" in binary is 00111001010. Consider what would happen if the leftmost bit were changed during transmission. The received character would be 10111001010 instead of 00111001010. The four parity bits will be checked by the receiver with the results:

parity bit 1 incorrect (bits 1, 3, 5, 7, 9, and 11 contain three 1's)

parity bit 2 correct (bits 2, 3, 6, 7, 10, and 11 contain two 1's)

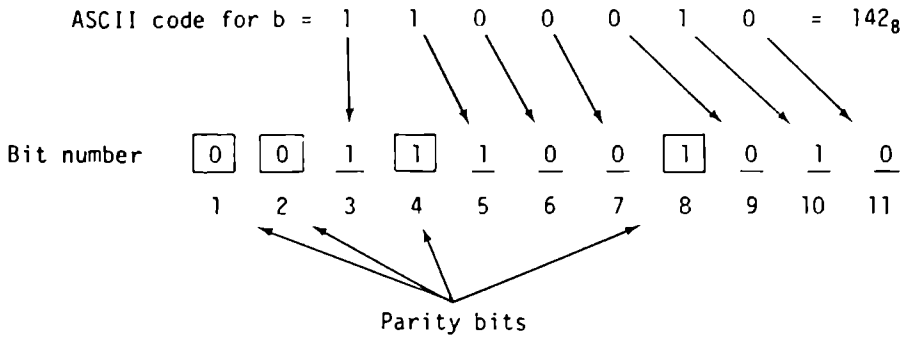


Fig. 2-11. Construction of the Hamming code for the letter "b" (ASCII code 1100010).

parity bit 4 correct (bits 4, 5, 6, and 7 contain two 1's)

parity bit 8 correct (bits 8, 9, 10, and 11 contain two 1's)

The total number of 1's in bits 1, 3, 5, 7, 9, and 11 should be an even number because even parity is being used. The incorrect bit must be one of the bits checked by parity bit 1—namely, bits 1, 3, 5, 7, 9, or 11. Because parity bit 2 is correct, we know that bits 2, 3, 6, 7, 10, and 11 are correct, so the error was not in bits 3, 7, or 11. That leaves 1, 5, and 9. Parity bit 4 is correct, meaning that bits 4, 5, 6, and 7 contain no errors. That narrows the choice down to 1 or 9. Parity bit 8 is also correct, so bit 9 is correct. Consequently, the incorrect bit must be bit 1. Because it was received as a 1, it must have been transmitted as a 0. In this manner, errors can be corrected.

A simple method for finding the incorrect bit is first to compute all the parity bits in the received character. If all are correct, there was no error (or more than one). Then add up all the incorrect parity bits, counting 1 for bit 1, 2 for bit 2, 4 for bit 4, and so on. The resulting sum is the position of the incorrect bit. For example, if parity bits 1 and 8 are correct, but 2 and 4 are incorrect, then bit 6 (2 + 4) has been inverted.

We will now briefly consider the problem of multiple errors. A character code is a list of all the characters producible by a certain device and their mappings onto the binary integers of a certain number of bits. Consider a pair of characters taken from some characters code. For this, and every other pair of characters, one can ask the question, "How many bits must be changed in character 1 to produce character 2?" If the two characters have the same bit pattern except for one bit, then only one bit must be changed in order to produce one from the other. If one consists of N ones and the other consists of N zeros, then N bits must be changed to produce one from the other. The minimum number of changes that can produce one character from the other is called the **distance** between the characters. A character code in which at least one pair of characters has a distance K, but no pair has a distance less than K, is called a distance K code.

If every possible combination of N bits is used in a certain code, it is a distance 1 code, since changing one bit in any character will still produce a valid character.

A code that uses N data bits and one parity bit to encode 2^N characters is a distance 2 code, since making one change to any character will give a parity error (i.e., a combination that is not a valid character) but making a second change will restore the correct parity. In general, a distance P code will detect up to P — 1 errors, since, by definition, it takes at least P changes to produce a valid character from another valid character. It is not possible to change a valid character into another valid character with only P — 1 changes.

2.3.5. Frequency-Dependent Codes

When storing or transmitting large amounts of text, it frequently pays to look for ways to compress the text into a smaller number of bits. The time needed to transmit a certain message is proportional to the number of bits in the message. By compacting the data to be sent, transmission times can be reduced. Furthermore, compacted data requires fewer bits to store.

One way of doing so is to eliminate the restriction that all character codes must be the same length. If the codes for common letters such as "e" and "t" were shorter than the codes for rare letters such as "q" and "x," the total number of bits needed to store or transmit the text would be decreased. Such a coding scheme is called a frequency-dependent code or **Huffman code** (Huffman, 1952).

This method can only be used if the probability of occurrence of each character is known at the time the code is designed. A code based on the probabilities for English text may not be optimum for PL/I programs, and it will be far from optimum for transmitting messages in Hungarian or Polish. Figure 2-12 gives a frequency distribution for English text.

As an example, we will construct a Huffman code for the digits 0, 1, 2, . . . , 9, based on the probabilities shown below.

Digit	0	1	2	3	4	5	6	7	8	9
Frequency	.20	.25	.15	.08	.07	.06	.05	.05	.05	.04

It should be emphasized that in order to use this encoding method for a particular application, the a priori frequencies of use for each character must be known. The code to be developed as an example is appropriate only for those transmissions where 0's represent about 20 percent of the characters, 1's 25 percent, and so on.

A 6.22	J 0.06	S 5.81
B 1.32	K 0.31	T 7.68
C 3.11	L 3.07	U 2.27
D 2.97	M 2.48	V 0.70
E 10.53	N 5.73	W 1.13
F 1.68	O 6.06	X 0.25
G 1.65	P 1.89	Y 1.07
H 3.63	Q 0.10	Z 0.06
I 6.14	R 5.87	space 18.21

Fig. 2-12. Percent distribution of letters and space for English text. Sample taken from first draft of this book (669,701 characters).

The first step in constructing the code is to write down the probability of each character under the character, as above. The order in which the characters are listed is unimportant and may be changed during the construction. Next, the two smallest probabilities are found and a new probability added to the table equal to the sum of the two smallest. The smallest probabilities are then marked with an asterisk so they will not be used again, and lines are drawn between the old and new ones. This process is shown in Fig. 2-13(a).

The process of finding the two smallest unmarked probabilities and creating a new one is then repeated. In Fig. 2-13(b), .05 and .05 are the two smallest. In Fig. 2-13(c), .06 and .07 are the two smallest. In Fig. 2-13(d), .08 and .09 are the two smallest. The fact that .08 is one of the original probabilities and .09 was generated from .04 and .05 is unimportant. The next step uses .08 and .09 to make .17.

Figure 2-14 shows the complete tree built up by repetition of this process until only one unmarked probability remains, whose value is 1.00. The characters have been rearranged for readability. To derive the code for a character, start at the root of the tree (1.00) and follow the path to the character, writing down a 0 for every left branch taken and a 1 for every right branch. For example, to get from the root to the character 6, first go left to .43 (i.e., do not go right to .57) and write down 0. Then go right to .23 and write down a 1, giving 01 so far. Then go right again, giving 011. Finally, go left, giving 0110. The complete code is shown below.

Digit	Huffman Code	Path from 1.00 to Character
0	00	.43, .20
1	10	.57, .25
2	110	.57, .32, .15
3	1110	.57, .32, .17, .08
4	0100	.43, .23, .13, .07
5	0101	.43, .23, .13, .06
6	0110	.43, .23, .10 .05
7	0111	.43, .23, .10 .05
8	11110	.57, .32, .17, .09, .05
9	11111	.57, .32, .17, .09, .04

The shortest codes, 00 and 10, are assigned to the two most common characters, whereas the two longest codes are assigned to relatively infrequently occurring characters. This situation, of course, is to be contrasted with the one where each character is assigned a four-bit code.

Characters 0, 1, and 2 together account for 60 percent of the characters to be transmitted and all three are shorter than four bits. If this character code is used for encoding numbers having the preceding distribution, 45 percent of the characters will require 2 bits, 15 percent will require 3 bits, 31 percent will require 4 bits, and 9 percent will require 5 bits. This averages to 3.04 bits per character. A list of 1,000,000 numbers would require only 3,040,000 bits using this code, whereas 4,000,000 bits would be needed using a four-bit code.

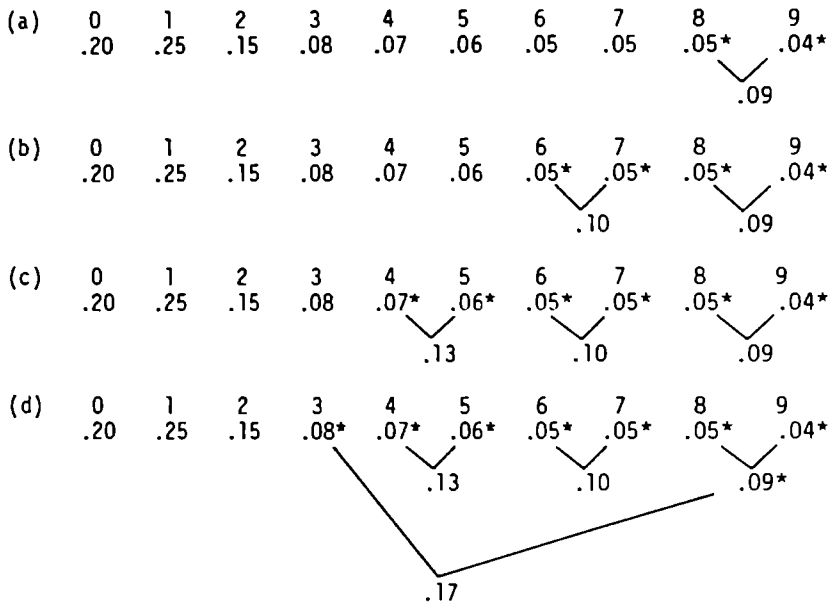


Fig. 2-13. Construction of a Huffman code for the digits 0–9 based upon the a priori relative frequencies given in the text.

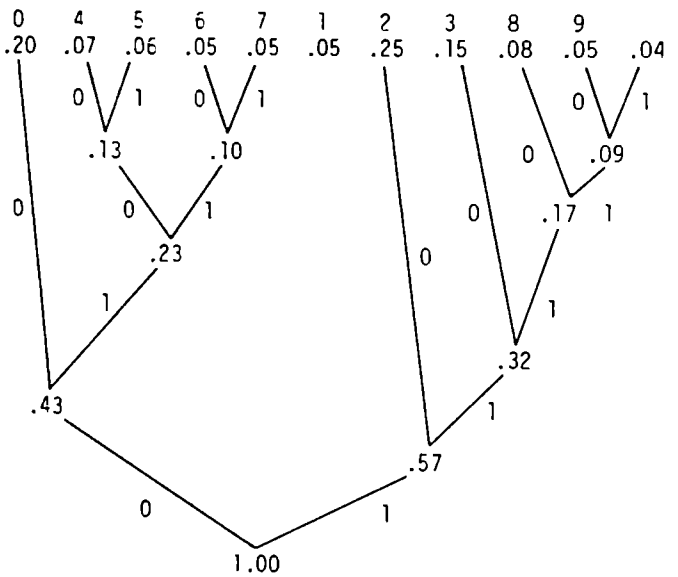


Fig. 2-14. The complete Huffman code tree begun in Fig. 2-13.

The character code generated by this algorithm is not unique. By interchanging the positions of 8 and 9 in Fig. 2-14, for example, we get 11111 for 8 and 11110 for 9, instead of the other way around. Furthermore, replacing all the 1's by 0's, and vice versa, also produces a new code. However, none of these transformations affects

the information density of the code, since the only thing that matters is how many bits each character is assigned, not the particular value of the code.

The decoding process for a Huffman-coded message received one bit at a time consists of examining the first bit to see if it is a valid code. If it is not, the first two bits are examined to see if they constitute a valid code. If not, the first three bits are checked, and so forth. Eventually a character is recognized, and the process is repeated, beginning with the first bit following the character.

Huffman codes have the disadvantage of being very sensitive to error. If one bit is received wrong, the receiving program will lose track of the character boundaries and become hopelessly lost. To improve the reliability, the Huffman-coded message could be divided into groups of N bits, with each group of N bits then encoded in a Hamming code.

2.4. TRANSFER OF INFORMATION

The preceding sections of this chapter have discussed the three major components of digital computer systems: processors, memory, and input/output equipment. These components need to communicate with one another. Processors need to fetch instructions and data from memories. Disks have to communicate with memories in order to put their information there. CPUs need to communicate with i/o processors in order to give them work to do.

Moreover, it is commonplace for a computer at one computer center to transmit data to a computer at another computer center. Networks of computers are becoming increasingly important in many applications and for many users. In the following sections we will discuss the issues and problems involved in the communication of information between computer components and interconnected computers.

2.4.1. Data Paths

A processor which controls i/o devices needs to be able to access them in order to start, stop, and communicate with them. Furthermore, the i/o devices need access to the main memory in order to read data from it and write data into it. Two possible organizations are depicted in Fig. 2-15. Figure 2-15(a) illustrates a separate physical data path for each possible transfer. Figure 2-15(b) shows the use of a single bus to connect all components. A **bus** consists of a number of parallel wires, with separate wires for data, addresses, synchronization, control, and, possibly, error detection and error correction.

In the direct connection organization, there is a dedicated cable for each pair of components that needs to communicate. Since each connection is permanently attached to two devices, a device receiving information along a particular data path need not check to see who the sender is; there is only one possibility. Consequently, no addresses are sent along with the data. This form of organization has the advantages of being straightforward and highly efficient, for many data transmissions can occur in parallel, along different data paths. It has the disadvantage of requiring many

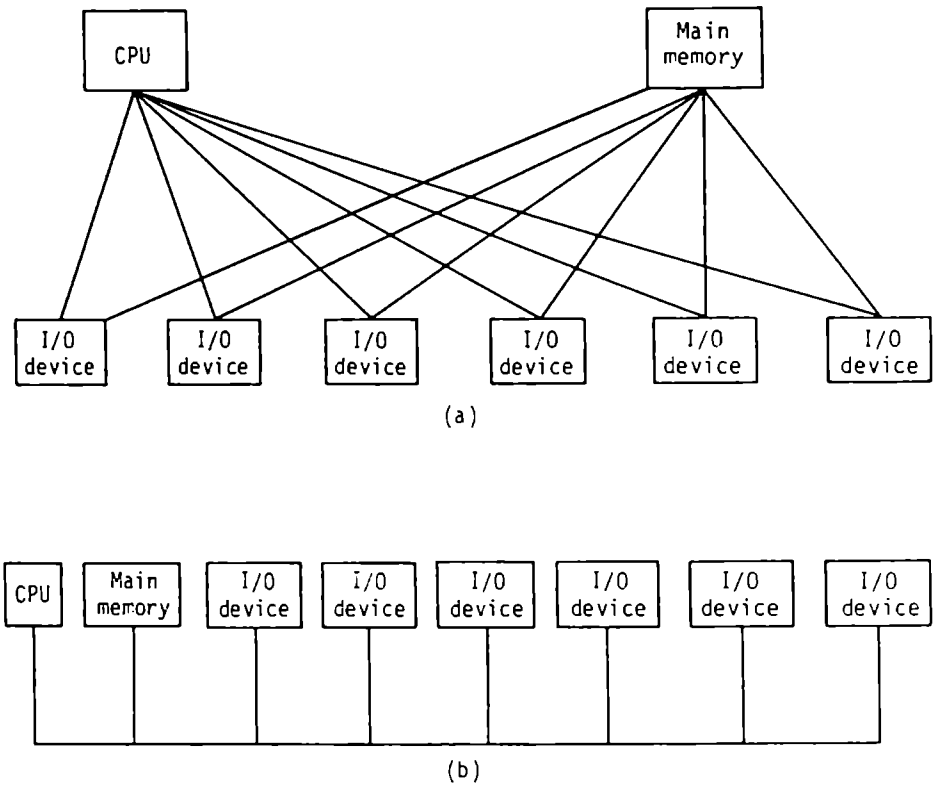


Fig. 2-15. Two possible systems organizations. (a) Individual data paths for each connection. (b) A single bus.

connections, many of which are idle most of the time, which is expensive and wasteful.

In the single bus organization, each device on the bus has an address. If the main memory also uses the bus for communication, each word or byte also has a bus address. When one device wants to communicate with another device, it puts the address of the device it wants to talk to on the address lines of the bus. The device addressed then sends an acknowledgement message back to the sender, signifying that it is ready. The communication can then proceed. Because only one message can be on the bus at any given time, there is no possibility of simultaneous data transfers between several pairs of devices, as in the direct connection organization. Consequently, a bus can be a bottleneck, causing a slowdown as devices wait for their turn to use it.

2.4.2. Telecommunications

It is frequently necessary for computers to communicate with i/o devices or other computers located some distance away. The terminals of a time-sharing system are often located many miles away from the computer, for example. At first, one might consider the possibility of simply stringing a wire between the computer and each

terminal, but the cost of doing so is almost always prohibitive. Moreover, in countries where the telephone company enjoys a monopoly, it is also illegal. As a result, it is usually necessary to employ the existing telephone network for communications purposes. This situation is perhaps unfortunate in that telephone networks were designed for transmitting the human voice, not digital data, but they can be used to transmit digital data if the number of bits per second is sufficiently low.

The maximum number of bits per second that can be transmitted over a given channel (telephone line, radio transmission, etc.) is a characteristic of the channel. The limitation is imposed by the noise level of the channel. If an attempt is made to transmit more bits per second than the channel is capable of transmitting, some information will be lost; that is, errors will occur.

There is, of course, no precise maximum bit rate for a channel in the sense that messages transmitted at, say, 1,204,364 bits per second will be received without error and messages transmitted at 1,204,365 bits per second will be hopelessly garbled. What happens is that as the bit rate increases, the error rate also increases. Above a certain error rate, a channel may be unacceptable. Telephone lines are rarely used at bit rates exceeding 4800 bits/sec. In contrast, most tape drives can transmit information to main memory at a rate exceeding 1,000,000 bits/sec. A computer with a 16-bit word and a memory cycle time of 1 μsec can transfer information to or from the memory at 16,000,000 bits/sec.

To transmit digital information over an analog line, each second is divided into N indivisible time intervals. During each interval, one or more bits can be transmitted. For example, if the computer could set the line to 1, 2, 3, or 4 volts during any time interval, those four voltages could be used to represent 00, 01, 10, or 11, thus allowing two bits to be sent per time interval. An N **baud** line is one in which the signal can change N times per second (i.e., N intervals per second). If each change has four possibilities, as in the preceding example, the transmission is called **dibit**, and the bit rate is twice the baud rate.

Modulation

Information entering or leaving a digital computer is in binary form. The voltage on an input line takes on only two values, as shown in Fig. 2-16(a). Two-level signals suffer considerable distortion when transmitted over a voice-grade telephone line, thereby leading to transmission errors. A pure sine wave signal at a frequency of 1000 to 2000 Hz, called a **carrier**, can be transmitted with relatively little distortion, however, and this fact is exploited as the basis of most telecommunications systems.

Because the pulsations of a sine wave are completely predictable, a pure sine wave transmits no information at all. However, by varying the amplitude, frequency, or phase, a sequence of 1's and 0's can be transmitted, as shown in Fig. 2-16. This process is called **modulation**.

In amplitude modulation [Fig. 2-16(b)], two different voltage levels are used, for 0 and 1, respectively. A person listening to digital data transmitted at a very low data rate would hear a loud noise for a 1 and no noise for a 0.

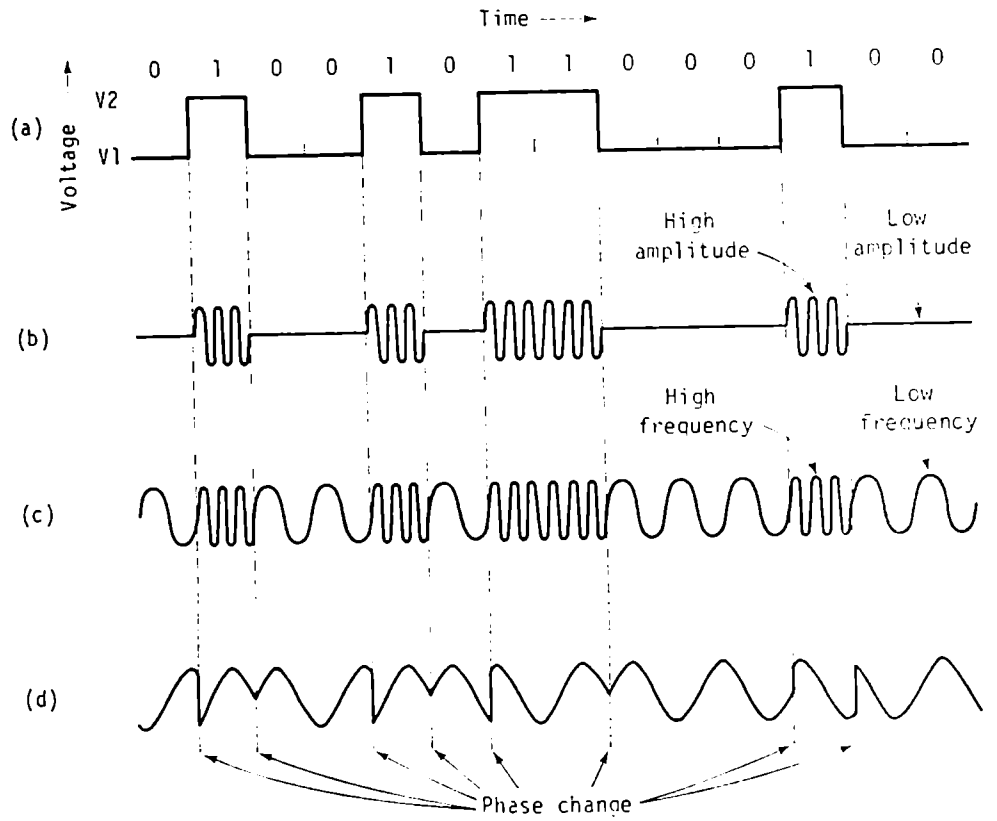


Fig. 2-16. Transmission of the binary number 01001011000100 over a telephone line bit by bit. (a) Two-level signal. (b) Amplitude modulation. (c) Frequency modulation. (d) Phase modulation.

In frequency modulation [Fig. 2-16(c)], the voltage level is constant, but the carrier frequency is different for 1 and 0. A person listening to frequency-modulated digital data would hear two tones, corresponding to 0 and 1.

In simple phase modulation [Fig. 2-16(d)], the amplitude and frequency do not change, but the phase of the carrier is reversed 180 degrees when the data switches from 0 to 1 or 1 to 0. In more sophisticated phase-modulated systems, at the start of each indivisible time interval, the phase of the carrier is abruptly shifted by 45, 135, 225, or 315 degrees, to allow two bits per time interval. Dibit phase encoding is one of the most commonly used methods, for reasons of cost and reliability.

If the data to be transmitted consists of a series of eight-bit characters, it would be desirable to have a connection capable of transmitting eight bits simultaneously—that is, eight pairs of wires. Since voice-grade telephone lines provide only one channel, the bits must be sent serially, one after another (or in groups of two if dibit modulation is being used). A device that accepts characters from a computer in the form of two-level signals, one bit at a time, and transmits the bits in groups of 1 or 2, in

amplitude, frequency, or phase-modulated form, is called a **modem** (MOdulator-DEModulator).

The transmitting modem sends the individual bits within one character at regularly spaced time intervals; for example, 1200 baud implies one signal change every 833 μsec. A second modem at the receiving end is used to convert a modulated carrier to a binary number. Since the bits arrive at the receiver at regularly spaced intervals, once the receiving modem has determined the start of the character, its clock tells it when to sample the line to read the values of the individual bits.

Asynchronous and Synchronous Transmission

There are two different methods of transmitting a series of characters. In **asynchronous transmission**, the time interval between two characters is not fixed, although the time interval between two consecutive bits within a character is fixed. For example, a person typing at a time-sharing terminal will not type at uniform speed, so the time interval between two successive characters will not be constant.

This raises the problem of how the receiver can recognize the first bit of a character. If the modulation methods of Fig. 2-16 are used, there is no way to distinguish between no data and a 0 bit. A character consisting entirely of 0's would be completely invisible. Furthermore, a character consisting of a 1 followed by seven 0's could not be distinguished from a character consisting of seven 0's followed by a 1, since the receiver would have no way of recognizing the start of the character.

In order to permit the receiver to recognize the start of a character, a start bit is transmitted directly before each character. To improve reliability, one or two stop bits are transmitted directly after each character. If the line is kept in the 1 state while no data is being transmitted, the start and stop bits are 0's, of course, so they can be recognized; and, conversely, if the line is in the 0 state while quiescent, the start and stop bits must be 1's. Between the start and stop bits, the data bits are transmitted at uniformly spaced time intervals. A timer in the receiving modem is started when the start bit arrives, thus allowing the modem to tell which bit is which. Asynchronous communication is illustrated in Fig. 2-17(a).

The most common bit rates used for asynchronous transmission are 110, 300, 600, 1200, and 2400 bits/sec. At 110 bits/sec, two stops are used, so a seven-bit ASCII character plus parity bit plus one start bit plus two stop bits gives 11 bits per character. Therefore 110 bits/sec corresponds to 10 characters/sec. The higher rates use only one stop bit.

In **synchronous** transmission, the need for non-information-carrying start and stop bits is eliminated. The result is to speed up data transmission. Synchronous communication often proceeds at bit rates of 4800 bits/sec, 9600 bits/sec, or even higher. In this method, once the modems have synchronized, they continue to send characters in order to remain in synchronization, even if there is no data to transmit. A special "idle" character is sent when there is no data. In ASCII, SYN is generally used. In synchronous transmission, unlike asynchronous transmission, the interval between two characters is always exactly the same.

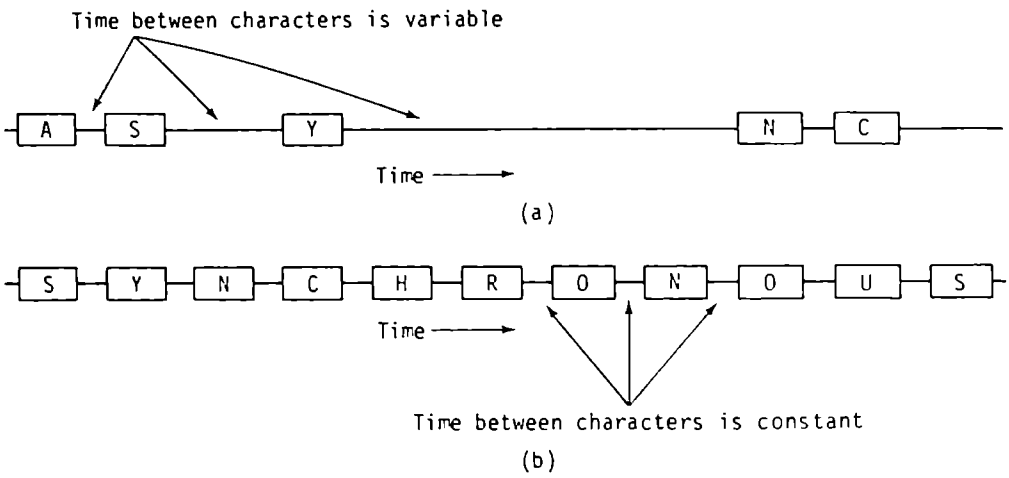


Fig. 2-17. Transmission modes. (a) Asynchronous transmission. (b) Synchronous transmission.

Synchronous transmission requires that the clocks in the transmitter and receiver remain synchronized for long periods of time, whereas asynchronous transmission does not, since the start of each character is explicitly indicated by a start bit. The length of time a transmission can be carried on without resynchronizing the receiver to the phase of the transmitter depends on the stability of the clocks. Typically, the clocks are stable enough to allow blocks of thousands of characters to be sent without having to resynchronize. These blocks sometimes use Hamming codes or other techniques to detect and correct transmission errors. Synchronous transmission is illustrated in Fig. 2-17(b).

Simplex, Half-Duplex, and Full-Duplex Transmission

Three modes of transmission are used for communications purposes: simplex, half duplex, and full duplex. A **simplex** line is only capable of transmitting data in one direction. The reason is not due to any property of the wires themselves but simply to the fact that one end has only a transmitter and the other end has only a receiver. This configuration is rarely used by computers because it provides no way for the receiver to transmit an acknowledgement signal to the sender, indicating that the message was received correctly. Radio and television broadcasting are examples of simplex transmission.

A **half-duplex** line can send and receive data in both directions, but not simultaneously. During any transmission, one modem is the transmitter and the other is the receiver. A common situation is for device A, acting as the transmitter, to send a series of characters to device B, acting as the receiver. Then A and B simultaneously switch roles, and B sends a message back to A specifying whether the characters were received without error or not. If there were no transmission errors, A and B switch roles again, and A sends the next message to B. If there were errors, A retransmits

the garbled message again. The "conversation" between the transmitter and the receiver about what to do next is called the **protocol**. The time required to switch a half-duplex line from one direction to the other may be many character times.

A **full-duplex** line can send and receive data in both directions simultaneously. Conceptually, a full-duplex line is equivalent to two simplex lines, one in each direction. Since two transmissions may be proceeding in parallel, one in each direction, a full-duplex line can transmit more information than a half-duplex line of the same data rate. Furthermore, full-duplex lines do not waste any time switching directions.

2.5. COMPUTER NETWORKS

A group of interconnected computers capable of exchanging information is called a **network**. Networks are organized as an interconnected series of **nodes**, each node consisting of one or more computers. Because the computers comprising the network can communicate with each other, the people using any of the computers can draw on the resources of any of the other computers in the network. This possibility offers some potential benefits.

First, the overall reliability of the network can be much higher than the reliability of any of its components. The ARPA network, which has dozens of nodes across the United States and Europe, has PDP-10s at MIT, the University of Utah, and Stanford. If the University of Utah PDP-10 is out of service for repairs one day, Utah programmers allowed to use the ARPA network can still have their programs run at MIT or Stanford. The network allows programmers to continue their work as usual, even though their local machine is not available.

Second, if the work load at one node becomes more than that node can handle, some of it can be farmed out to another node with a lighter work load. Load sharing can be of particular benefit if the network spans several time zones. At 10 A.M. Eastern Standard Time, many of the programmers working at MIT have arrived at work and are busy utilizing the local computers. At that moment it is only 7 A.M. in California, and probably there is little or no work to be done at Stanford. Similarly, at 4 P.M. Pacific Standard Time, there is a lot of activity at Stanford, but most of the programmers at MIT have long since gone home. A network allows programmers at one computer center to take advantage of unused computer facilities at another computer center, thus making more economical use of the system. From this point of view, an international network spanning all 24 time zones would be ideal.

Third, a network allows people to take advantage of unusual computers or other facilities located at a great distance. Scientists all over the United States can take advantage of the enormous computing power of the ILLIAC IV, located in California. Other resources that can be used via a network are large data bases (for example, a computerized medical library), and unusual i/o devices.

Figure 2-18 shows two alternative networks topologies. Each circle represents one node. In the star configuration, Fig. 2-18(a), any node can communicate with any other node by sending a message to the central node, D. The message is then sent

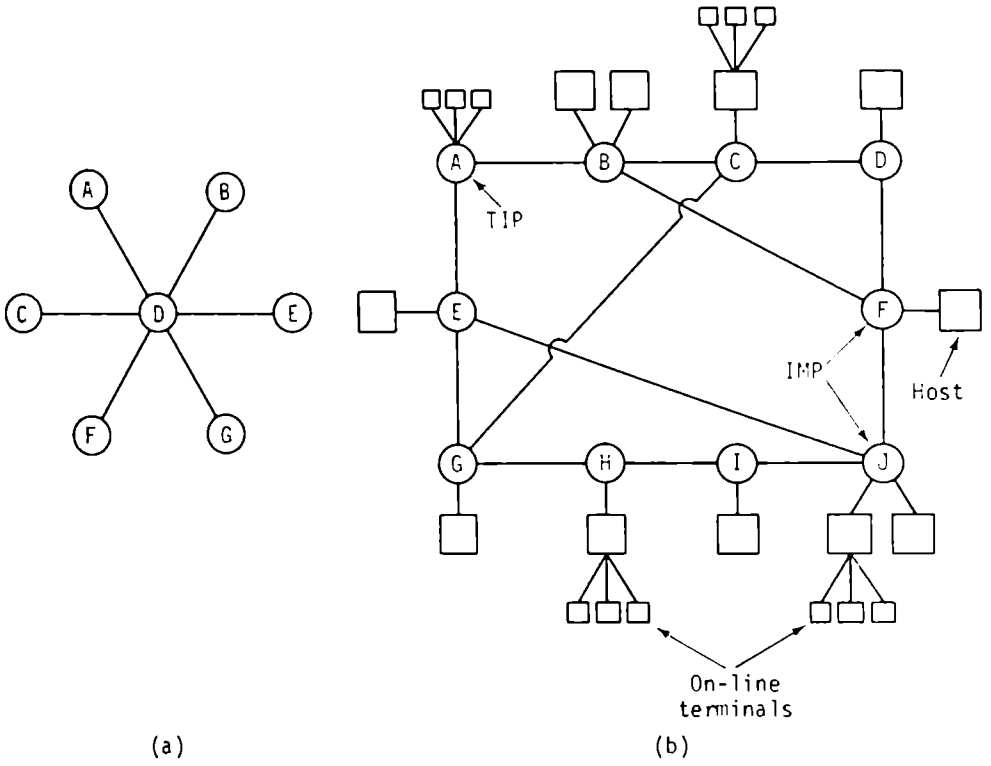


Fig. 2-18. Network configurations. (a) Star network. (b) Distributed network.

to the destination. The star configuration has the disadvantage that if the central node is out of service, the network ceases to function.

An alternative topology is the distributed network, an example of which is shown in Fig. 2-18(b). In a distributed network, there is no central node to which all the nodes are connected. The decision as to which nodes should be connected to which other nodes is based on such factors as distance between the nodes, amount of message traffic expected between the nodes, and existence of appropriate communications facilities between the nodes. In a distributed network, alternate paths from every node to every other node can be, and usually are, provided. For example, if a power failure at H completely removes it from the network, G can still communicate with I via the path G-E-J-I. If neither H nor E is functioning, G can use the path G-C-D-F-J-I to communicate with I.

The ARPA network is a highly sophisticated network, with nodes at many universities and research institutes. Each node of the ARPA network consists of a minicomputer to which may be connected one or more other large computers, called **hosts**. The minicomputers are called IMPs (Interface Message Processors) and TIPs (Terminal IMPs) and are responsible for the communications aspect of the network. The TIPs are parasite nodes. They allow on-line terminals to connect directly to the

network without using a host computer. Information is moved around the network from node to node in messages (or **packets**) of up to about 8000 bits. Each message contains a code specifying its ultimate destination.

If an on-line user at node H in Fig. 2-18(b) wants to communicate with the main computer at node D, he types his message into the main computer at H. The message is then passed to the IMP at H. The IMP puts the message into a queue in its own memory. A little later, the message is retrieved and sent to the IMP at G. The IMP at G examines the message to see if it is intended for the host computer at G, and, upon discovering that the message is for D, puts the message into a queue in its memory, and continues whatever it was doing when the message arrived. When that message's turn arrives, the message will be sent to the C IMP, where it will be examined to see if its destination is C and then put in a queue. Finally, it will be sent to the IMP at D, which will recognize that the message is for the host computer at D and give it to the host computer.

The ARPA network is an example of a store and forward network, so named because messages are stored at each node along the route and forwarded to the next node at a later time. In general, messages cannot be forwarded immediately because the needed transmission line may be in use forwarding another message, and there may already be several other messages in the queue patiently waiting for their turns to be forwarded along that same transmission line.

Networks are a new and evolving field of research with many unsolved problems. One problem is how to permit different kinds of computers in the network in such a way as to allow programs written for one kind to be run on another kind. Another problem involves the sharing of data between computers with different data formats, character codes, and so on. Still another problem concerns determining how to route messages to their destination most efficiently. For example, in Fig. 2-18(b) the route from A to C via A-B-C, may look shorter than A-E-J-F-D-C, but if the queue at B is full and the queues at E, J, F, and D are all empty, the latter route may be faster.

A further problem is deciding how to share the work load among the nodes. For example, if someone at E has a very large program that he wants to run using a very large data base at D as input, should the data be sent to E or should the program be sent to D? Perhaps both program and data should be sent to F if there is a light work load there.

Yet another problem involves accounting. If a user at H has a program at E and a file at D sent to F for some computing, how do you keep track of who pays for what?

2.6. DISTRIBUTED COMPUTERS

A variation on the idea of a computer network is being made possible by the development of inexpensive CPUs that can fit on a single integrated circuit chip. Complete CPUs contained on chips smaller than a pack of matches are already commercially available, and new and more powerful ones are being announced fre-

quently. The amount of memory that can be squeezed onto the same chip as the CPU is increasing rapidly every year. Foster (1972b) has predicted that single chips containing complete computers and their memories will be on sale for under 10 dollars by 1980.

The widespread availability of powerful single chip computers at a nominal cost will have enormous impact on computer organization. Highly parallel machines may become very attractive. Special purpose processors, such as i/o processors and disk controllers, may be replaced by single chip CPUs programmed to perform the appropriate function. If a manufacturer need only design and produce one kind of processor, thus replacing a variety of distinct special purpose processors, the total cost will be reduced due to the efficiency of mass production.

It is even possible that the concept of a central processor will become obsolete. A large computer may be equipped with dozens, or hundreds, or maybe even thousands of identical single chip (or sub chip) CPUs, each performing a specialized function. Some of these CPUs might be microprogrammed to compile programs (with each CPU specializing in only one language), run compiled programs (again specializing the machine to the language), handle communication with on-line users, manage large data bases, monitor system performance, run hardware diagnostics, maintain a display for the console operator, or drive specific i/o devices.

The central processor concept might be replaced by the idea of a distributed processor. Today's computer networks may be the forerunners of tomorrow's distributed computers. The difference between a network and distributed computer is principally a question of the amount of interaction between the nodes, and the size, location and capabilities of the nodes. If all the components of a system are relatively small, dependent, and geographically close together, we may regard them as part of the same computer. If, on the other hand, they are powerful and self-sufficient, we may regard them as distinct computers. Hybrid forms of many varieties are likely to become common in the years ahead. The whole area of multiple processor systems is a rapidly emerging area of research and development, and you are encouraged to investigate the current state of the art.

PROBLEMS

1. What is the function of each of the following?
   a. CPU
   b. memory
   c. i/o processor
   d. data path
2. What is the purpose of step 2 in the list of Section 2.1.1? What would happen if this step were omitted?
3. A certain computation is highly sequential in nature—that is, each step depends on the one preceding it. Would an array processor or a pipeline processor be more appropriate for this computation? Explain.

4. To compete with the newly invented printing press, a certain medieval monastery decided to mass produce handwritten paperback books by assembling a vast number of scribes in a huge hall. The head monk would then call out the first word of the book to be produced, and all the scribes would copy it down. Then the head monk would call out the second word, and all the scribes would copy it down, too. This process was repeated until the entire book had been read aloud and copied. Which of the parallel processor systems of Fig. 2-3 does this method most resemble?

5. A standard punched card has 80 columns of 12 positions. How large a character set can be used if it is required that every character be represented by a single column with no more than three punches?

6. Make an estimate of how many characters, including spaces, a typical book contains. How many bits are needed to encode a book in ASCII? How many magnetic tapes are needed to store a library of 10^6 books?

7. Make an estimate of the maximum storage capacity of the human brain, using the following assumptions. All memory is coded in the form of DNA molecules. A DNA molecule is a linear sequence of nucleotides. There are four nucleotides: A, C, G, and T. How many bits can be encoded in a nucleotide? From the average weight of a nucleotide, roughly 10^{-20} gram, and an average brain weight of 1500 grams, deduce the bit capacity of the brain for this encoding form. (*Note*: This calculation provides only an upper limit to the information capacity. The actual bit capacity is much lower.)

8. Which of the following memories are possible? Which are reasonable? Explain.

	Number of Bits in the MAR	Number of Cells in Memory	Cell Size in Bits
a.	10	1024	8
b.	10	1024	12
c.	9	1024	10
d.	11	1024	10
e.	10	10	1024
f.	1024	10	10

9. Sociologists can get three possible answers to a typical survey question like "Do you believe in Santa Claus?"—namely, yes, no, and no opinion. The Sociomagnetic Manufacturing Company has decided to build a computer to process survey data. This computer has a trinary memory—that is, a byte consists of eight trits, each trit holding a 0, 1, or 2. How many trits are needed to hold a six-bit number? Give an expression for the number of trits needed to hold N bits.

10. A certain computer can be equipped with up to 131,072 words of main memory. Why would a manufacturer choose such a peculiar number rather than an easy-to-remember number like 100,000 or 125,000?

11. Consider a computer memory consisting of 2^{18} 64-bit words. There are four numeric data types: binary integers, binary floating-point numbers, decimal integers, and decimal floating-point numbers. The CPU has only one ADD instruction and uses the memory's metabits to determine the type of add needed. How many metabits are

needed per word? How many bits of memory are required in all, data bits plus meta-bits, for the entire memory?

12. How could a computer memory be organized so as to allow individual words to be designated as read only, read/write, or execute only?

13. Why do some computers have special i/o processors?

14. An ordinary magnetic tape has 1600 bytes/inch. If a company invented a new kind of tape unit, which could read and write at density of 3200 bytes/inch, what advantages would you expect?

15. A disk has N concentric cylinders, 0 to N-1. If the arm is positioned at cylinder K,
    a. what is the maximum inward move (cylinder 0 is innermost)?
    b. what is the maximum outward move?
    c. what is the average move, assuming equal probabilities for the next cylinder?
    d. evaluate the answer to part (c) for the three cases, K = 0, K = N-1, and K = N/2, in the limit $N \gg 1$.

16. The transfer rate between a CPU and its associated memory is orders of magnitude higher than the mechanical i/o transfer rate. How can this imbalance cause inefficiencies?

17. How fast can you read? Convert that to bits per second and compare with a card reader that can read 1200 80 column cards per minute. How much faster is the card reader than you?

18. Compute the data rate of the human eye based on the following information. The visual field consists of about 10^6 elements. The time resolution is about 100 msec. Any element can be reduced to a superposition of three colors, each of which has 64 intensities.

19. Decode the following ASCII text.

    1001001 0100000 1001100 1001111 1010110 1000101 0100000
    1011001 1001111 1010101

20. Devise a Hamming code for the digits 0 to 9.

21. Devise a distance 2 code for the digits 0 to 9.

22. In a Hamming code, some bits are "wasted" in the sense that they are used for checking and not information. What is the percentage of "wasted" bits for messages whose total length (data + check bits) is $2^N - 1$? Evaluate this expression for values of N from 3 to 10.

23. If a Hamming code for ASCII characters has a parity bit appended that checks the parity of the entire character, what is the distance of the resulting code?

24. Transmission errors on telephone lines often occur in bursts rather than individually. Since the basic Hamming code method can only correct single errors within a character, it is of no help if noise bursts garble N consecutive bits. Devise a method of transmitting ASCII text in blocks of 1000 characters over a line where noise bursts may garble as many as 100 consecutive bits. (*Hint:* Think about the order in which you should transmit the bits.)

25. Work out the Huffman code for English text based on Fig. 2-12. How many bits would

be needed, on the average, to transmit 1 million characters? How many bits would be needed if the ASCII code were used?

26. When even-parity ASCII text is transmitted asynchronously at a rate of 10 characters/sec over a 110-bit/sec line, what percentage of the received bits actually contain data (as opposed to start, stop, and parity bits)?

27. A certain asynchronous ASCII terminal does parity checking and prints at 60 characters/sec. Which speed line should it use: 110, 300, 600, or 1200 bits/sec?

28. The Hi-Fi Modem Company has just designed a new frequency-modulation modem that uses 16 frequencies instead of just 2. Each second is divided into N equal time intervals, each of which contains one of the 16 possible tones. How many bits per second can this modem transmit, using synchronous transmission?

29. How many characters/sec (seven bits + parity) can be transmitted over a 2400-bit/sec line in each of the following modes?
    a. synchronous
    b. asynchronous (one start bit and one stop bit)

30. What information *must* be present in the message packets of a distributed store-and-forward network in addition to the data? What information *might* also be present as well?

31. Two computer science students are having an argument about networks. The first student claims that if message traffic increases beyond a certain amount, the IMPs should be provided with more memory. The second student claims that if the message traffic is too high for the system, the IMPs should be replaced by faster IMPs. Under what conditions is each proposal reasonable? (*Hint*: Consider the bit rate of the transmission lines and the calculations involved in using very sophisticated error-correcting codes as potential bottlenecks.)

32. A certain university is planning to buy computers for all its campuses, to be connected in a network. Each of the computer manufacturers bidding for the contract is urging the planning group to buy the same kind of host computer for each node in the network (namely, its own brand). However, the computing requirements of the various campuses differ considerably, so the planning group is considering buying an assortment of different computers. What are the advantages and disadvantages of a homogeneous network (identical hosts) compared to a heterogeneous network (a variety of hosts)?

33. Write a procedure HAMMING (ASCII, ENCODED) that converts seven-bit ASCII characters to 11-bit characters using a Hamming code.

34. Write a function DISTANCE (CODE, N, K) that takes an N-element array, CODE, as input. CODE contains N characters of K bits each. The function should return the distance of the character set.

35. Write a simulator to analyze the performance of a moving head disk. Read in requests for service from cards. Each request specifies the cylinder to be read from or written onto, the number of bytes to be transferred, and the time the request is made. The simplest algorithm is the one that handles requests on a first-come, first-served basis. However, if the input rate is high, other requests may come in while the first request is being carried out, so a collection of requests for service may be pending. A good

algorithm can choose the next request to be serviced so as to minimize seek time. The following algorithm is a good one.

Maintain lists of pending disk requests, one per disk cylinder and a variable, D, that keeps track of the direction of arm motion, +1 for outward and −1 for inward. Whenever a disk request for cylinder N finishes, check to see if there is a request pending for cylinder N + D. If so, carry it out. If not, check cylinder N − 2D, N − 3D, etc. When you get to the innermost cylinder, 0, set D = −D to switch directions. Similarly, switch directions at the outermost cylinder, K − 1, and begin moving inward to look for service requests.

Note that this is the algorithm used by elevators, with servicing a request being picking up or discharging a passenger. If the disk is heavily loaded, it may always be possible to find a request within one or two cylinders, thus greatly reducing the average seek time over the first-come, first-served algorithm. The simulation should determine (at least)

a. the average number of cylinders moved per seek
b. the average elapsed time between a request coming in and being satisfied

as a function of two parameters: the average number of requests per second and the average size of a request. Use the parameters of your computer's disk; but if you cannot determine them, use the PDP-11s.

number of cylinders = 203

tracks per cylinder = 2

bytes per track = 6144

rotation time = 40 msec

transfer rate = 1 track/40 msec

seek time to move n cylinders = $6 + 2n$ $(0 < n \leq 8)$ msec

$16 + 3n/4$ $(9 \leq n \leq 24)$ msec

$26 + n/3$ $(n \geq 25)$ msec

For simplicity, assume that no request is longer than a track. Be sure to include seek time, rotational delay, and transfer time in your model.

36. Write a simulator for the network of Fig. 2-18(b) in order to study network performance. For each node, maintain a list of all messages waiting in the IMP to be transmitted. For each message, keep track of the number of bits yet to be transmitted and the ultimate destination, but not the data. Maintain a clock, and advance it in units of 1 sec. Every time the clock ticks, check each IMP to see if there is a message to be forwarded; and if so, subtract N bits from it. (The transmission rate, N, is a parameter of the simulation.) When the bit count at the sending node is reduced, the bit count at the receiving node must be correspondingly increased. When a message arrives at its final destination, remove it from the simulation. Note that you only need to keep track of how many bits are in which queue, and you do not have to keep track of the messages themselves.

The input data consists of a sequence of cards, each containing four items: the starting node, the destination node, the message length, and the time the message is created. The cards should be sorted in order of increasing time. The message itself is not needed.

The main program of the simulator will advance the clock, call a procedure to see if any new messages have just arrived, and call procedures to check each node for messages and simulate their transmissions. A new message arriving at any node, either one originating there or one just passing through, should be put at the end of the queue.

As output, the simulator should provide the following information (as a bare minimum):

a. average input rate of messages (in bits per second).
b. average number of bits transmitted per second.
c. average queue length in each IMP in bits.
d. average percent delay per message (the actual transmission time divided by the time it would have taken had the network been empty).
e. average utilization of each transmission line.

Study how these and similar parameters vary with system loading (average input rate) and transmission rate (N).

3 THE CONVENTIONAL MACHINE LEVEL

This chapter introduces the conventional machine level (level 2) and discusses many aspects of its organization, instructions, and data. Before we begin, it is worth pointing out why level 2 is being described before level 1 (the microprogramming level), upon which it is built. The primary reason is that a substantial part of the level 2 organization and instruction set is also present at level 3, the level to which most assemblers and compilers translate. To begin with level 3, however, would be to start with an unnecessarily complicated machine. Also, many of the ideas presented in this chapter are also applicable to the other levels, but they can be explained more easily in the context of level 2.

Another reason is that the conventional machine level was historically the first one, and the microprogramming level was later developed as a technique for implementing it. A thorough knowledge of the conventional machine level will help in understanding why the microprogramming level has developed the way it has.

3.1. EXAMPLES OF THE CONVENTIONAL MACHINE LEVEL

Rather than attempting to define rigorously what the conventional machine level is (which is probably impossible, anyway), we will introduce this level by means of three examples. The next three sections are devoted to examining the conventional machine level of three families of well-known, commercially available computers: the IBM 370s, the CDC Cyber 70s, and the DEC PDP-11s. These same three families of

computers can also be examined at their other levels, and we will do so in subsequent chapters. The purpose of choosing three existing computer families to study is to show how the ideas discussed here can be applied to the "real world." These machines will be compared and contrasted in many ways, and they will serve as running examples throughout the rest of the book.

You should not draw the conclusion that the remainder of the book is about programming the 370s, Cyber 70s, or PDP-11s. These machines will be used to illustrate the idea of designing a computer as a series of levels. Various features of their respective organizations will be examined, and some information about programming them will be introduced where necessary. However, you should keep in mind the central, unifying idea that computers can be designed in a structured way. The technique of building a computer as a series of levels is a powerful structuring technique. An understanding of some of the details and idiosyncracies of the three machines is necessary in order to understand the various levels, but try to relate the details to the overall structure and do not wallow in them.

3.1.1. IBM System/360 and System/370

The IBM System/360 and System/370 constitute a series of several dozen different models, spanning a wide range in performance and applications. From the level 2 programmer's point of view, the various models of both the 360 series and 370 series have nearly identical properties and organization, although from the level 1 programmer's standpoint, they are radically different. Henceforth we will refer only to the System/370 series, or just 370 for short, although the comments apply to the System/360 series as well.

An IBM 370 consists of one or more CPUs, one or more i/o processors, a main memory, and various i/o devices, as shown in Fig. 3-1. There are three kinds of i/o processors, called multiplexer channels, block multiplexer channels, and selector channels, the first type being used with low-speed i/o devices, such as card readers, printers, and card punches, and the other two being used with high-speed i/o devices, such as disks, drums, and tapes. All the processors in the system have access to main memory for reading and writing. The channels have a few internal registers but no main memory of their own.

The smallest addressable unit in the main memory is the **byte**, consisting of eight bits. Each byte has a unique address. There are a maximum of 2^{24} distinct addresses; therefore the bytes are numbered 0, 1, 2, 3, 4, . . . , N − 1, where N is the number of bytes of memory, up to a maximum of $N = 2^{24}$. Two consecutive bytes form a half word, 4 consecutive bytes form a word, and 8 consecutive bytes form a double word. Words are more important than half words or double words, and the 370 is often regarded as having a 32-bit word. The address of a half word, word, or double word is the address of its lowest-numbered byte, which, on some older models, must be an integral multiple of 2, 4, or 8, respectively. Figure 3-2 illustrates the addressing structure of the 370 memory.

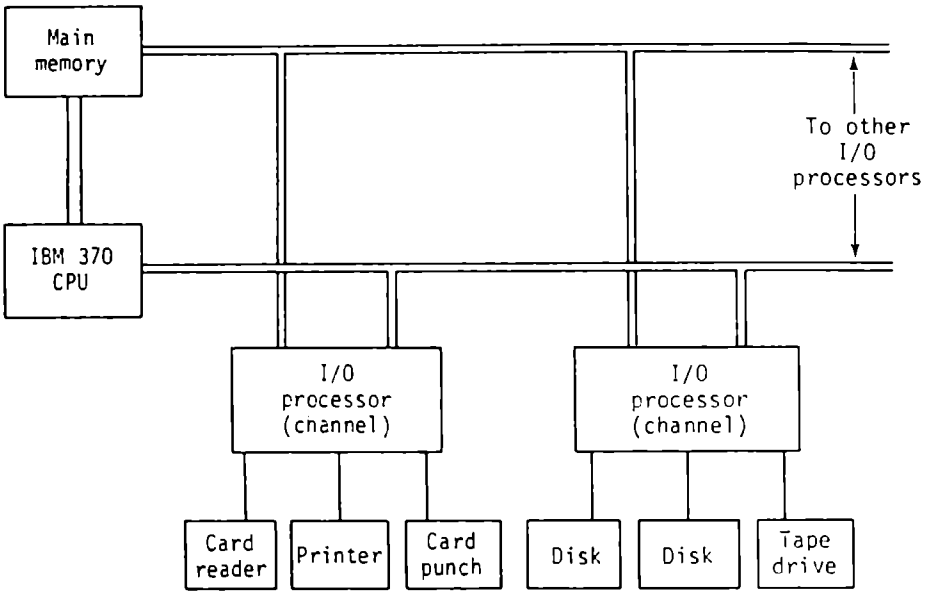


Fig. 3-1. Organization of an IBM 370 computer with one CPU and two i/o processors (channels).

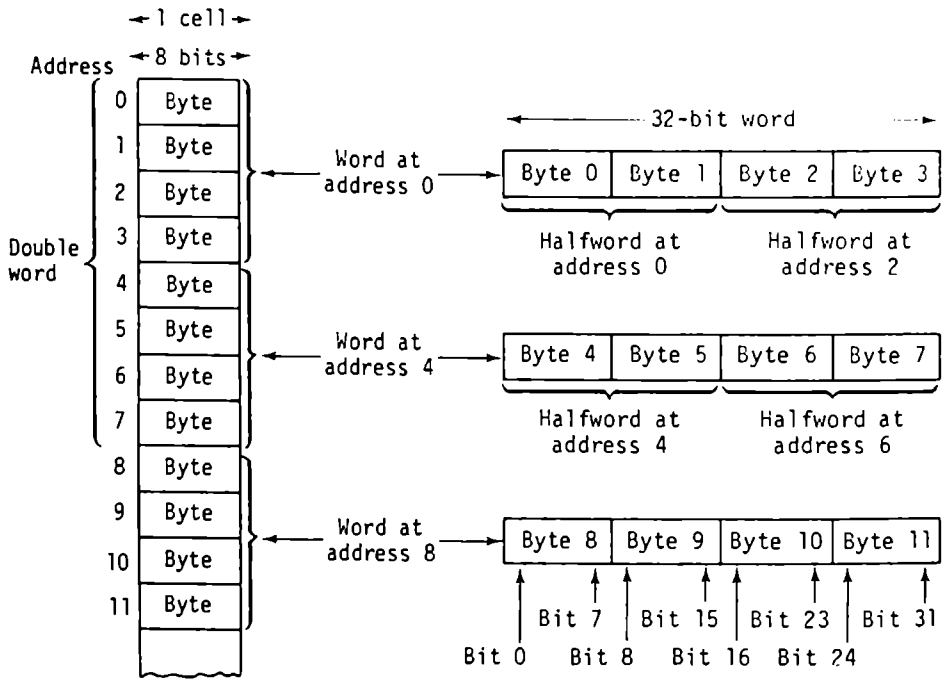


Fig. 3-2. Addressing structure of the IBM 370's main memory.

The IBM 370 has instructions for fetching bytes, half words, words, and double words. Note that each byte is part of a half word, a word, and a double word. For example, byte 7 is the low-order (rightmost) byte of the half word at 6, the word at 4, and the double word at 0.

The 370 has a special format for storing packed decimal numbers. Four bits are needed to represent a digit in the range 0 to 9, so two decimal digits can be packed into one eight-bit byte. In this format, a byte may contain any number from 0 to 99. If used to store numbers in pure binary form, a byte can hold any number between 0 and 255, and seven bits are sufficient to hold all the numbers from 0 to 99. The packed decimal format does not use memory optimally.

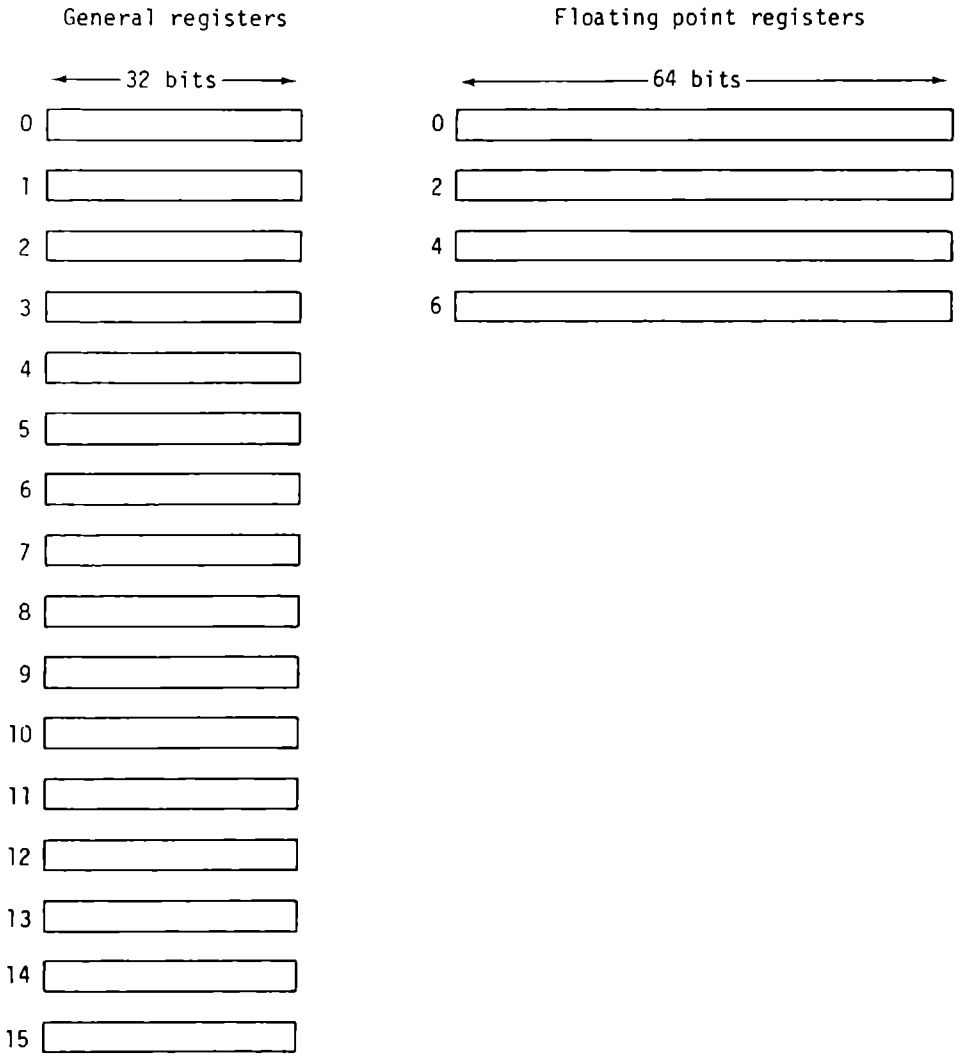


Fig. 3-3. General registers and floating point registers on the 370.

RR format

xxxx	Branching and status switching 0000xxxx		Fixed-point fullword and logical 0001xxxx		Floating-point long 0010xxxx		Floating-point short 0011xxxx	
0000			LPR	Load positive	LPDR	Load positive	LPER	Load positive
0001			LNR	Load negative	LNDR	Load negative	LNER	Load negative
0010			LTR	Load and test	LTDR	Load and test	LTER	Load and test
0011			LCR	Load complement	LCDR	Load complement	LCER	Load complement
0100	SPM	Set program mask	NR	And	HDR	Halve	HER	Halve
0101	BALR	Branch and link	CLR	Compare logical	LRDR	Load rounded (E to L)	LRER	Load rounded (L to SH)
0110	BCTR	Branch on count	OR	Or	MXR	Multiply (E)	AXR	Add normalized (E)
0111	BCR	Branch/condition	XR	Exclusive or	MXDR	Multiply (L to E)	SXR	Subtract normalized (E)
1000	SSK	Set storage key	LR	Load	LDR	Load	LER	Load
1001	ISK	Insert key	CR	Compare	CDR	Compare	CER	Compare
1010	SVC	Supervisor call	AR	Add	ADR	Add N	AUR	Add N
1011			SR	Subtract	SDR	Subtract N	SER	Subtract N
1100			MR	Multiply	MDR	Multiply	MER	Multiply
1101			DR	Divide	DDR	Divide	DER	Divide
1110	MVCL	Move long	ALR	Add logical	AWR	Add U	AUR	Add U
1111	CLCL	Compare logical long	SLR	Subtract logical	SWR	Subtract U	SUR	Subtract U

RX format

xxxx	Fixed-point halfword and branching 0100xxxx		Fixed-point fullword and logical 0101xxxx		Floating-point long 0110xxxx		Floating-point short 0111xxxx	
0000	STH	Store	ST	Store	STD	Store	STE	Store
0001	LA	Load address						
0010	STC	Store character						
0011	IC	Insert character						
0100	EX	Execute	N	And				
0101	BAL	Branch and link	CL	Compare logical				
0110	BCT	Branch on count	O	Or				
0111	BC	Branch/condition	X	Exclusive or	MXD	Multiply (L to E)		
1000	LH	Load	L	Load	LD	Load	LE	Load
1001	CH	Compare	C	Compare	CD	Compare	CE	Compare
1010	AH	Add	A	Add	AD	Add N	AE	Add N
1011	SH	Subtract	S	Subtract	SD	Subtract N	SE	Subtract N
1100	MH	Multiply	M	Multiply	MD	Multiply	ME	Multiply
1101			D	Divide	DD	Divide	DE	Divide
1110	CVD	Convert-decimal	AL	Add logical	AW	Add U	AU	Add U
1111	CVB	Convert-binary	SL	Subtract logical	SW	Subtract U	SU	Subtract U

Fig. 3-4. The IBM 370 instruction set. The instructions marked * are actually groups of instructions with a 16-bit opcode. The abbreviation for each instruction is given before the name.

RS, SI, S format

xxxx	Branching status switching and shifting 1000xxxx		Fixed-point logical and input/output 1001xxxx		1010xxxx		1011xxxx	
0000	SSM	Set system mask	STM	Store multiple				
0001			TM	Test under mask			LRA	Load real address (15 instructions)*
0010	LPSW	Load PSW	MVI	Move				
0011		diagnose	TS	Test and set				
0100	WRD	Write direct	NI	And				
0101	RDD	Read direct	CLI	Compare logical				
0110	BXH	Branch/high	OI	Or			STCTL	Store control
0111	BXLE	Branch/low-equal	XI	Exclusive or			LCTL	Load control
1000	SRL	Shift right SL	LM	Load multiple				
1001	SLL	Shift left SL						
1010	SRA	Shift right single					CS	Compare and swap
1011	SLA	Shift left single					CDS	Compare D and swap
1100	SRDL	Shift right DL	SIO	Start I/O *	STNSM	Store then and mask		
1101	SLDL	Shift left DL	TIO	Test I/O *	STOSM	Store then or mask	CLM	Comp chars masked
1110	SRDA	Shift right double	HIO	Halt I/O *	SIGP	Signal processor	STCM	Store chars masked
1111	SLDA	Shift left double	TCH	Test channel	MC	Monitor call	ICM	Insert chars masked

SS format

xxxx	1100xxxx	Logical 1101xxxx		1110xxxx	Decimal 1111xxxx	
0000					SRP	Shift and round
0001		MVN	Move numerics		MVO	Move with offset
0010		MVC	Move		PACK	Pack
0011		MVZ	Move zones		UNPK	Unpack
0100		NC	And			
0101		CLC	Compare logical			
0110		OC	Or			
0111		XC	Exclusive or			
1000					ZAP	Zero and add
1001					CP	Compare
1010					AP	Add
1011					SP	Subtract
1100		TR	Translate		MP	Multiply
1101		TRT	Translate and test		DP	Divide
1110		ED	Edit			
1111		EDMK	Edit and mark			

Note: N = Normalized
SL = Single logical
DL = Double logical
U = Unnormalized
E = Extended
L = Long
SH = Short

Fig. 3-4 (cont.)

Packed decimal numbers do have certain advantages over binary numbers, however. Data input to the computer or output from the computer is in decimal notation, because people use decimal numbers. When binary numbers are used internally, the input must be converted from decimal to binary, processed in binary, and then reconverted from binary to decimal. If the amount of computation is small, the CPU may spend most of its time performing conversions. If the amount of computation is large, the faster speed of the binary arithmetic instructions makes the conversions worthwhile. Packed decimal numbers are widely used in business applications.

A program at the conventional machine level on a 370 has access to 20 high-speed CPU registers used for performing arithmetic and logical operations, as well as for storing intermediate results. Of this number, 16 are general-purpose registers of length 32 bits, numbered from 0 to 15. The other 4 registers, numbered 0, 2, 4, and 6, are 64 bits long and are used for floating-point arithmetic. The 370 hardware also contains a number of other registers, such as an instruction register, program status word, MAR, and MBR, but they are only accessible at the microprogramming level. Figure 3-3 illustrates the registers used by conventional machine level programs. There are also 16 control registers, but they are only used by the operating system and will not concern us further.

The 370 level 2 instructions are 16, 32, or 48 bits in length and may be located at any even address. Nearly all instructions contain an eight-bit operation code, specifying which instruction is to be carried out. The remaining bits are used to specify where the data for the instruction is located—in registers, memory, or both. The 370 has some general-purpose instructions used in almost all programs, some instructions intended primarily for scientific calculations, and some instructions primarily useful for commercial applications. The level 2 instruction set contains 183 instructions. A list of most of these instructions is given in Fig. 3-4.

3.1.2. CDC 6000, Cyber 70, and Cyber 170

The CDC 6400, 6500, 6600, Cyber 72, Cyber 73, Cyber 74, and Cyber 170 series computers are a group of powerful, large-scale computers particularly intended for large numerical calculations and other applications where an appreciable amount of computing power is needed. From the programmer's point of view (as opposed to the engineer's), the Cyber 72 is nearly identical with the 6400, the Cyber 73 is nearly identical with the 6500, and the Cyber 74 is nearly identical with the 6600. Furthermore, the 6500 is merely a designation for a 6400 with two CPUs, and a 6600 is essentially a highly parallel and thus faster 6400. The Cyber 170s are improved versions of the Cyber 70s. Unlike the IBM 360s and 370s, which together span a performance range of a factor of several hundred, these CDC machines span a performance range of a factor of about 10. For convenience, we will refer to the entire group as the Cyber 70.

A CDC Cyber 70 consists of one or more CPUs, between 10 and 20 i/o processors, a main memory, possibly an ECS (Extended Core Storage), and various i/o devices,

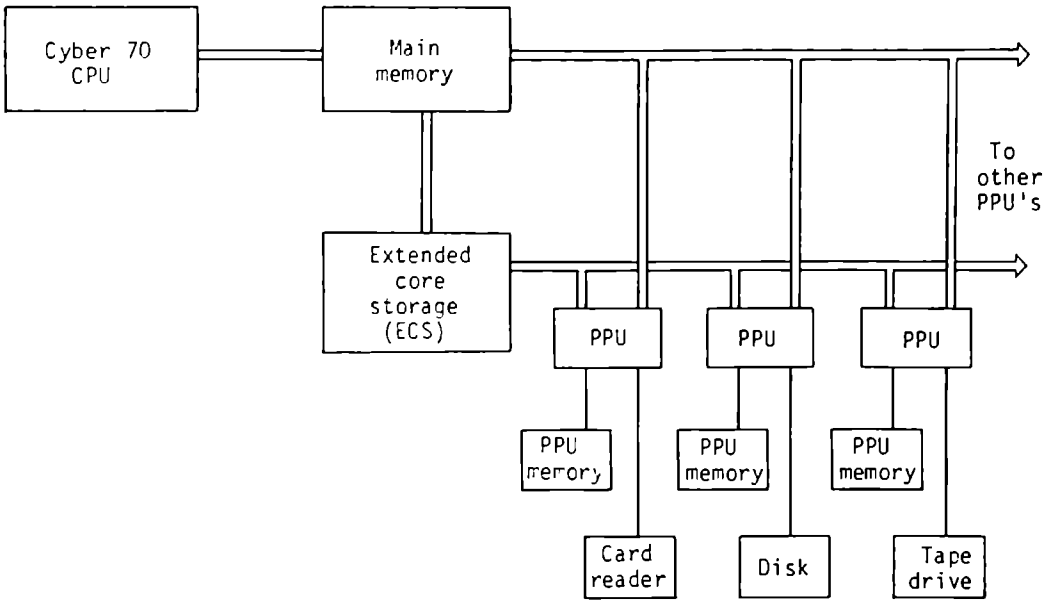


Fig. 3-5. Organization of the CDC Cyber 70 computer. The PPU's are i/o processors.

as shown in Fig. 3-5. CDC calls its i/o processors "PPUs" (Peripheral Processing Units), whereas IBM calls its i/o processors "channels."

Unlike the IBM channels, which have only a few words of memory each and a very limited instruction repertoire, each CDC PPU is a complete, general-purpose, digital computer with its own memory of 4096 12-bit words and an instruction set of 66 instructions. Furthermore, the IBM 370 CPU can stop or start a channel at any time, whereas the CDC Cyber 70 CPU has no way to force an uncooperative PPU to start, stop, or even pay attention at all. Each PPU has an area of main memory assigned to it. The CPU communicates with the PPUs, and the PPUs communicate with each other by putting messages in these areas. This communication is used to implement level 3 machine instructions.

If a PPU is loaded with a program containing a bug, it may cease to examine its communication area and, in effect, stop being part of the system. Such a rogue PPU can cause considerable damage, including erasing the main memory and all the disks. The only way to stop a rogue PPU is to bring the entire system to a halt from the operator's console and then do a "dead start," which consists of loading a short program into one PPU from a group of toggle switches on the back panel. In contrast, the IBM 370 CPU can forcibly stop any channel at any time.

The Cyber 70 main memory consists of up to 2^{17} (131, 072) words of 60 bits each. Unlike the IBM 370, which can read memory in four different-sized units—namely, bytes, half words, words, and double words—the Cyber 70 instructions always fetch exactly one 60-bit word on a READ and always store exactly one 60-bit

word on a WRITE, except for transfers between main memory and ECS, which consist of an integral number of 60-bit words. Figure 3-6 illustrates the Cyber 70 memory.

The Cyber 70 conventional machine level has 24 registers that are directly accessible to the CPU programmer. The group of 24 is divided into three subgroups of

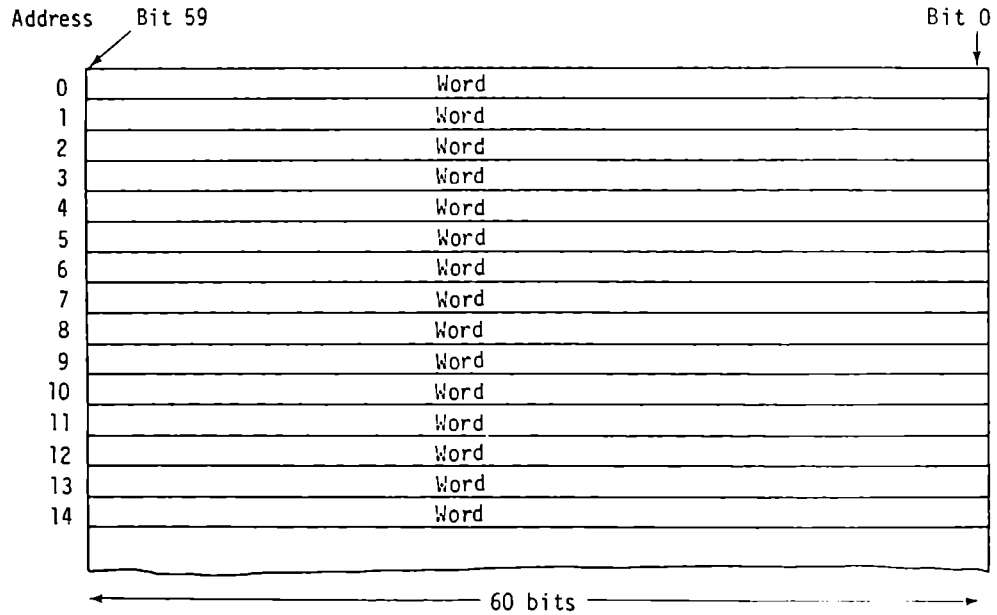


Fig. 3-6. Addressing structure of the Cyber 70's main memory.

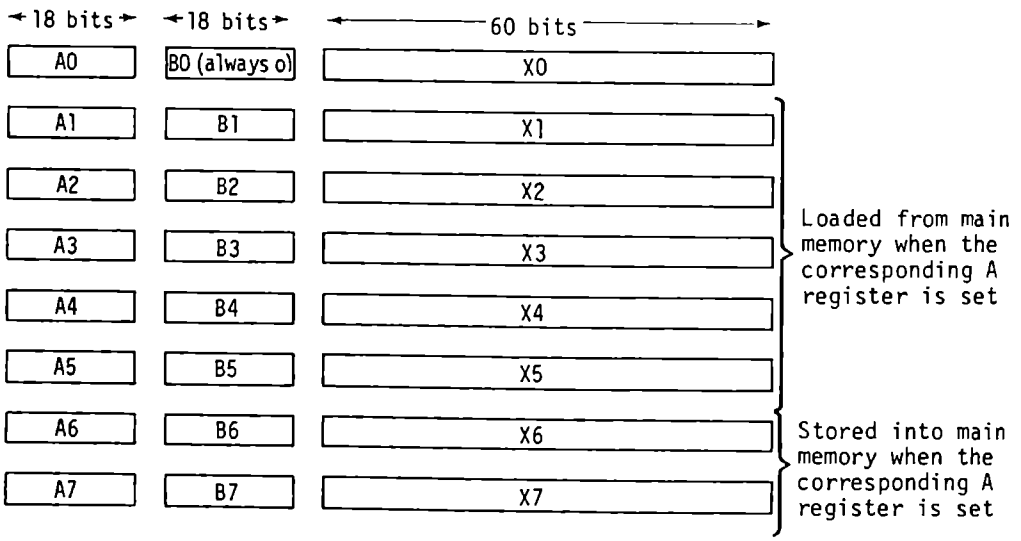


Fig. 3-7. Cyber 70 registers.

eight registers apiece, the A, B, and X registers, as shown in Fig. 3-7. The A registers are 18 bits long and are called address registers. The B registers are also 18 bits long and are called increment registers, although index registers would have been a better choice. The X registers are 60 bits long and are called operand registers. The registers are named

A0, A1, A2, A3, A4, A5, A6, A7
B0, B1, B2, B3, B4, B5, B6, B7
X0, X1, X2, X3, X4, X5, X6, X7

The A registers (except A0) are really MARs, and the X registers, except X0, are really MBRs. When a memory address is put into A1, the contents of that address are automatically loaded in X1 about a microsecond later. Similarly, loading A2 to A5 with an address causes X2 to X5 to be loaded with the contents of the memory word in the corresponding A register. When an address is loaded into A6 or A7, the corresponding X register is stored into memory. Setting A0 does not initiate a memory reference. Registers A0 and X0 are used for transferring information between main memory and ECS, as well as for holding intermediate results. You should keep these properties of the A and X registers in mind in all subsequent discussions of the Cyber 70.

The maximum number of memory words on a Cyber 70 is 2^{17}. An 18-bit B register can hold a signed integer in the range $-(2^{17} - 1)$ to $2^{17} - 1$. Register B0 has the property of always being 0. There is no way to alter it, either accidently or intentionally. A store into B0 is simply ignored. The fact that B0 is always 0 has numerous uses. For example, there is an instruction to add two B registers together and put the sum in a third B register. If one of the registers to be added is B0, this instruction becomes "move B register," thus eliminating the need for a special instruction.

The CPU hardware also contains seven registers accessible only to the PPUs. These registers include the program counter, plus registers concerned with relocation, protection, and trap handling. In addition, several internal registers are used for temporary results during instruction execution—for example, a 96-bit register used by the floating-point multiplication instruction.

The Cyber 70 has three instruction lengths: 15, 30, and 60 bits. Instructions have a 6-or 9-bit operation code. The 30-bit instructions each specify a memory address in the low-order 18 bits. Instructions may not be split across word boundaries. Figure 3-8 shows the Cyber 70 instructions.

3.1.3. DEC PDP-11

The DEC PDP-11 series consists of a number of small-to medium-sized computers. Due to their short (16-bit) word length, they are often called **minicomputers**. The PDP-11s are widely used in such applications as communications, industrial process control, scientific experiment monitoring and data collection, interactive computer graphics, and education.

A PDP-11 consists of a CPU, a main memory, and various i/o devices, as illustrated in Fig. 3-9. Unlike the larger and more expensive 370 and Cyber 70, the PDP-11 has no i/o processors. All components of the PDP-11 are connected by a single bus

```
00XXX  PS   Error exit or program stop

0100N  RJ   Return jump to N
011jN  RE   Read ECS
012jN  WE   Write ECS
013jN  XJ   Central exchange jump

021XN  JP   Jump to (Bi) + N

030jN  ZR   Jump to N if (Xj) = 0
031jN  NZ   Jump to N if (Xj) ≠ 0
032jN  PL   Jump to N if (Xj) positive
033jN  NG   Jump to N if (Xj) negative
034jN  IR   Jump to N if (Xj) in range
035jN  OR   Jump to N if (Xj) out of range
036jN  DF   Jump to N if (Xj) definite
037jN  ID   Jump to N if (Xj) indefinite

04ijN  EQ   Jump to N if (Bi) = (Bj)
05ijN  NE   Jump to N if (Bi) ≠ (Bj)
06ijN  GE   Jump to N if (Bi) ≥ (Bj)
07ijN  LT   Jump to N if (Bi) < (Bj)

10ijX  BXi  Move (Xj) to Xi
11ijk  BXi  AND of (Xj) and (Xk) to Xi
12ijk  BXi  OR of (Xj) and (Xk) to Xi
13ijk  BXi  EXCLUSIVE OR of (Xj) and (Xk) to Xi
14iXk  BXi  Move 1's complement of (Xk) to Xi
15ijk  BXi  AND of (Xj) and compl (Xk) to Xi
16ijk  BXi  OR of (Xj) and compl (Xk) to Xi
17ijk  BXi  EX OR of (Xj) and compl (Xk) to Xi

20ijk  LXi  Left shift (Xi) by jk
21ijk  AXi  Right shift (Xi) by jk
22ijk  LXi  Left shift (Xk) by (Bj) to Xi
23ijk  AXi  Right shift (Xk) by (Bj) to Xi
24ijk  NXi  Normalize (Xk) to Xi and Bj
25ijk  ZXi  Round and normalize (Xk) to Xi and Bj
26ijk  UXi  Unpack (Xk) to Xi and Bj
27ijk  PXi  Pack Xi from (Xk) and (Bj)

30ijk  FXi  Floating add (Xj) and (Xk) to Xi
31ijk  FXi  Float subtract (Xk) from (Xj) to Xi
32ijk  DXi  Floating DP add (Xj) from (Xk) to Xi
33ijk  DXi  Float DP subtract (Xj) from (Xk) to Xi
34ijk  RXi  Rounded add (Xj) and (Xk) to Xi
35ijk  RXi  Rounded subtract (Xj) from (Xk) to Xi
36ijk  IXi  Integer add (Xj) and (Xk) to Xi
37ijk  IXi  Integer subtract (Xj) from (Xk) to Xi

40ijk  FXi  Floating multiply (Xj) and (Xk) to Xi
41ijk  RXi  Rounded multiply (Xj) and (Xk) to Xi
42ijk  DXi  DP or integer multiply to Xi
43ijk  MXi  Form mask in Xi of jk bits
44ijk  FXi  Floating divide (Xj) by (Xk) to Xi
45ijk  RXi  Rounded divide (Xj) by (Xk) to Xi
46000  NO   No operation
464jN  IM   Move indirect
465NQ  DM   Move direct
466NQ  CC   Compared collated
467NQ  CU   Compared uncollated
47iXk  CXi  Count the 1's in (Xk) to Xi

50ijN  SAi  Set Ai to (Aj) + N
51ijN  SAi  Set Ai to (Bj) + N
52ijN  SAi  Set Ai to (Xj) + N
53ijk  SAi  Set Ai to (Xj) + (Bk)
54ijk  SAi  Set Ai to (Aj) + (Bk)
55ijk  SAi  Set Ai to (Aj) - (Bk)
56ijk  SAi  Set Ai to (Bj) + (Bk)
57ijk  SAi  Set Ai to (Bj) - (Bk)

60ijN  SBi  Set Bi to (Aj) + N
61ijN  SBi  Set Bi to (Bj) + N
62ijN  SBi  Set Bi to (Xj) + N
63ijk  SBi  Set Bi to (Xj) + (Bk)
64ijk  SBi  Set Bi to (Aj) + (Bk)
65ijk  SBi  Set Bi to (Aj) - (Bk)
66ijk  SBi  Set Bi to (Bj) + (Bk)
67ijk  SBi  Set Bi to (Bj) - (Bk)

70ijN  SXi  Set Xi to (Aj) + N
71ijN  SXi  Set Xi to (Bj) + N
72ijN  SXi  Set Xi to (Xj) + N
73ijk  SXi  Set Xi to (Xj) + (Bk)
74ijk  SXi  Set Xi to (Aj) + (Bk)
75ijk  SXi  Set Xi to (Aj) - (Bk)
76ijk  SXi  Set Xi to (Bj) + (Bk)
77ijk  SXi  Set Xi to (Bj) - (Bk)

Notes: i, j, and k are 3-bit fields;
       N is an 18-bit address field;
       X is a 3-bit field that is not used;
       Q is 30-bit field with 4 subfields;

       all numbers are in octal;
       (register) means the contents of
                  the register
```

Fig. 3-8. The Cyber 70 instruction set.

called a UNIBUS. The UNIBUS is used for transmitting information from one component to another; for example, the CPU can read information from memory, the disk can store information into memory, or a CRT display can refresh its screen directly from the disk. At any instant in time, only one transfer may be proceeding—although the UNIBUS can be switched from one pair of devices to another, in a time interval on the order of a microsecond.

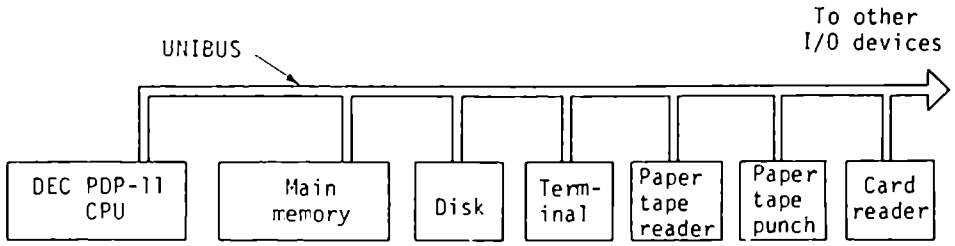


Fig. 3-9. Organization of a DEC PDP-11 computer.

The presence of only one communication path in the system, although conceptually simple and inexpensive, has some disadvantages. Consider, for example, the situation where the CPU is running while information is being read into the memory from the disk at a rate of one word per N CPU instructions. Most of the time the UNIBUS will be used to transmit information from the memory to the CPU, as shown in Fig. 3-10(a); but about once every N instructions the UNIBUS will be needed to transfer a word from the disk to memory, as shown in Fig. 3-10(b). When this situation occurs, the UNIBUS will be taken away from the CPU, and the disk-to-memory transfer will proceed. The CPU simply waits until the word has been transferred and the UNIBUS is then available again.

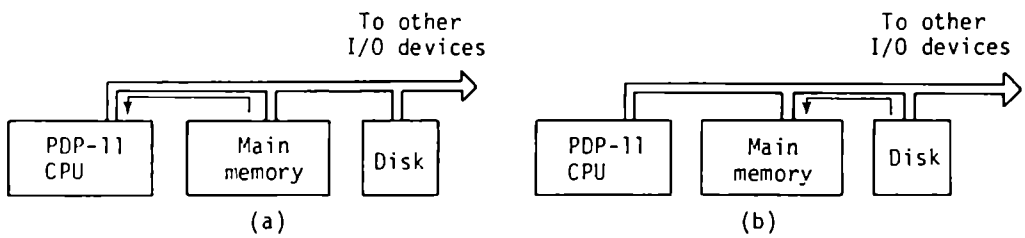


Fig. 3-10. (a) The UNIBUS being used to transfer data from main memory to the CPU; (b) The UNIBUS being used to transfer data from the disk to main memory.

This process, called **cycle stealing**, slows the CPU down by a little, because the CPU is idle while the disk is using the UNIBUS to communicate with the memory. Associated with each device is a priority. When two or more devices request use of the UNIBUS simultaneously, the one with the highest priority gets it and the other devices must wait. The CPU priority can be set by the program.

The smallest addressable unit in the PDP-11's memory is a byte, consisting of eight bits. Two bytes make a 16-bit word. Although the basic addressable unit (cell) on the PDP-11 is the same as on the 370, the PDP-11 word length is half that of the 370. The PDP-11 memory is illustrated in Fig. 3-11. Note that DEC, like CDC and unlike IBM, numbers its bits from right to left, so that bit N carries a weight of 2^N

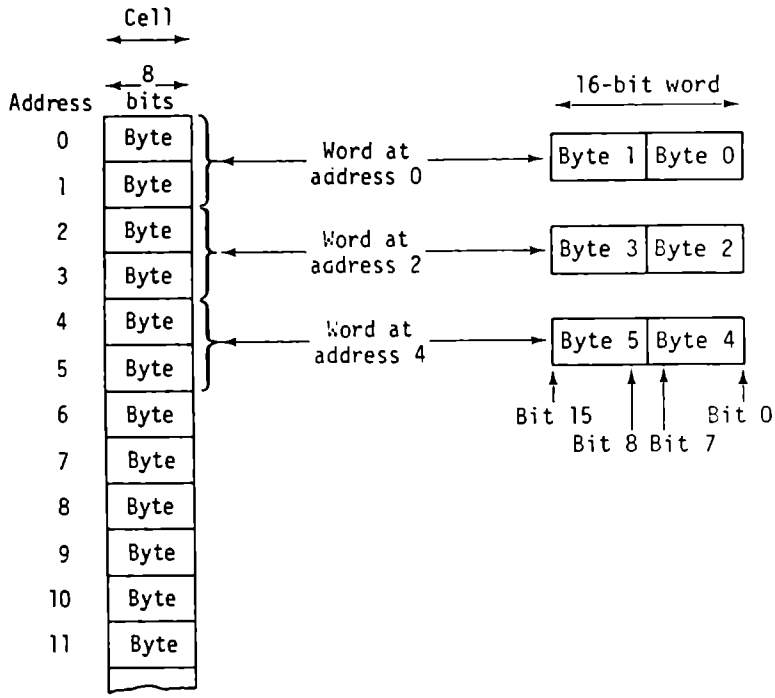


Fig. 3-11. Addressing structure of the PDP-11's main memory.

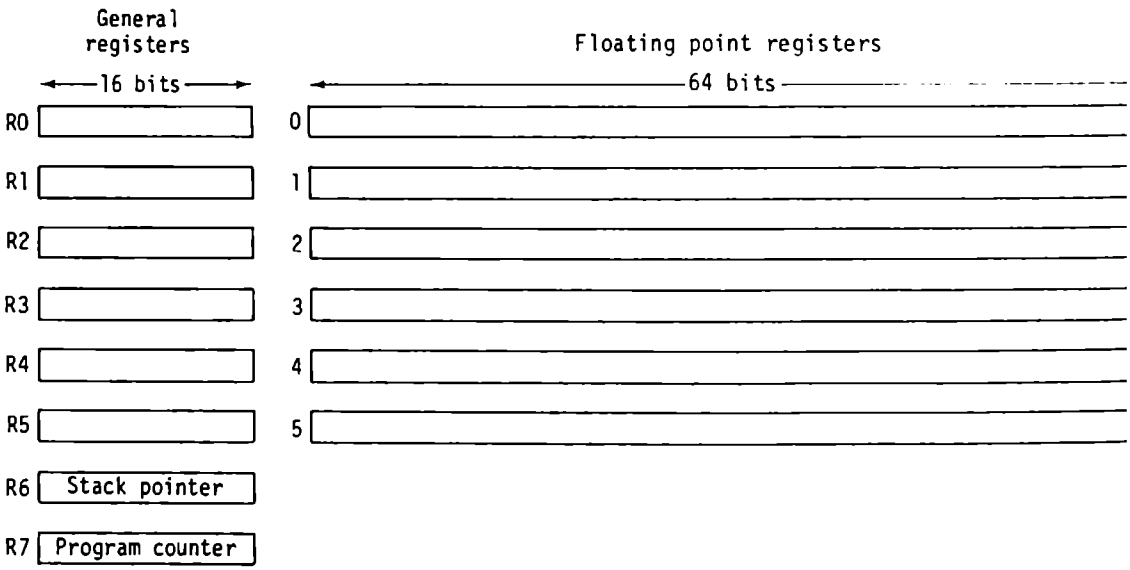


Fig. 3-12. PDP-11 registers.

Group 1, 4-bit opcodes. Format: xxxxssssssdddddd

xxxx		
0000	---	See groups 2 and 3
0001	MOV	Move
0010	CMP	Compare
0011	BIT	Bit test
0100	BIC	Bit clear
0101	BIS	Bit set
0110	ADD	Add
0111	---	See group 10
1000	---	See groups 11, 12, and 13
1001	MOVB	Move byte
1010	CMPB	Compare byte
1011	BITB	Bit test byte
1100	BICB	Bit clear byte
1101	BISB	Bit set byte
1110	SUB	Subtract
1111	---	(Floating point instructions)

Group 2, 8-bit opcodes. Format: 00000xxxkkkkkkkk

xxx		
000	---	See group 4
001	BR	Branch
010	BNE	Branch not equal
011	BEQ	Branch equal
100	BGE	Branch greater than or equal
101	BLT	Branch less than
110	BGT	Branch greater than
111	BLE	Branch less than or equal

Group 3, 7-bit opcodes. Format: 00001xxrrrdddddd

xx		
00	JSR	Jump to subroutine
01	---	See group 5
10	---	See group 6
11	---	Spare

Group 4, 10-bit opcodes. Format: 00000000xxdddddd

xx		
00	---	See group 7
01	JMP	Jump
10	---	See group 8
11	SWAB	Swap bytes

Group 5, 10-bit opcodes. Format: 0000101xxxdddddd

xxx		
000	CLR	Clear
001	COM	Complement (1's complement)
010	INC	Increment
011	DEC	Decrement
100	NEG	Negate (2's complement)
101	ADC	Add carry
110	SBC	Subtract carry
111	TST	Test

Group 6, 10-bit opcodes. Format: 0000110xxxdddddd

xxx		
000	ROR	Rotate right 1 bit
001	ROL	Rotate left 1 bit
010	ASR	Arithmetic shift right 1 bit
011	ASL	Arithmetic shift left 1 bit
100	(MARK)	Clean up stack. dddddd = count
101	(MFPI)	Move from previous instruction space
110	(MTPI)	Move to previous instruction space
111	(SXT)	Sign extend

Notes: ssssss specifies a source;
dddddd specifies a destination;
rrr specifies a register;

Group 7, 12-bit opcodes. Format: 0000000000xxcccc

xx		
00	---	See group 9
01	---	Spare
10	CCC	Clear condition codes
11	SCC	Set condition codes

Group 8, 13-bit opcodes. Format: 0000000010xxxrrr

xxx		
000	RTS	Return from subroutine
011	(SPL)	Set priority level

Group 9, 16-bit opcodes. Format: 0000000000000xxx

xxx		
000	HALT	Halt
001	WAIT	Wait
010	RTI	Return from interrupt
011	BPT	Breakpoint
100	IOT	I/O trap
101	RESET	Reset
110	(RTT)	Return from trap

Group 10, 7-bit opcodes. Format: 0111xxxrrrdddddd

xxx		
000	(MUL)	Multiply
001	(DIV)	Divide
010	(ASH)	Arithmetic shift
011	(ASHC)	Arithmetic shift combined
100	(XOR)	Exclusive or
111	(SOB)	Subtract one and branch

Group 11, 8-bit opcodes. Format: 10000xxxkkkkkkkk

xxx		
000	BPL	Branch on plus
001	BMI	Branch on minus
010	BHI	Branch high
011	BLOS	Branch low or same
100	BVC	Branch on overflow clear
101	BVS	Branch on overflow set
110	BCC	Branch on carry clear
111	BCS	Branch on carry set

Group 12,10-bit opcodes. Format: 1000101xxxdddddd

xxx		
000	CLRB	Clear byte
001	COMB	Complement byte
010	INCB	Increment byte
011	DECB	Decrement byte
100	NEGB	Negate byte
101	ADCB	Add carry byte
110	SBCB	Subtract carry byte
111	TSTB	Test byte

Group 13, 10-bit opcodes. Format: 1000110xxxdddddd

xxx		
000	RORB	Rotate byte right 1 bit
001	ROLB	Rotate byte left 1 bit
010	ASRB	Shift byte right 1 bit
011	ASLB	Shift byte left 1 bit
100	----	Spare
101	(MFPD)	Move from previous data space
110	(MTPD)	Move to previous data space
111	----	Spare

x specifies opcode bits;
kkkkkkkk specifies an offset or constant;
cccc specifies condition code bits

Fig. 3-13. The PDP-11 instruction set. The parenthesized instructions are not present on all models. Opcodes not listed are spares. The 46 optional floating point instructions are not shown.

in an integer. Furthermore, on the PDP-11, byte 0 is to the right of byte 1, whereas on the 370 the reverse is true. A comparison of Fig. 3-2 and 3-11 will make this point clearer.

The PDP-11 level 2 machine has eight general-purpose registers of 16 bits each and (optionally) six 64-bit floating-point registers. The PDP-11 registers are illustrated in Fig. 3-12. General registers 0 to 5 are used for a variety of purposes, but registers 6 and 7 have specific functions. Register 6 is a stack pointer (to be explained in Section 3.3.7). Register 7 is the program counter. It always contains the address of the next instruction word to be fetched. Each time a word containing an instruction (or part of an instruction) is fetched from memory, the program counter is automatically incremented by two. The increment is two instead of one because a word contains two bytes, and consecutive words have addresses differing by two. For example, 100 and 102 are addresses of consecutive memory words. Word addresses are always even.

Instructions are 16, 32, and 48 bits, although the formats are completely different from the 370's instructions. Figure 3-13 shows the conventional machine level instructions for the PDP-11/45. The instruction sets on the other models are slightly different.

3.2. INSTRUCTION FORMATS

A program consists of a sequence of instructions, each one specifying some particular action. Part of the instruction, called the operation code, or **opcode** for short, tells what action is to be performed. Many instructions contain, or specify the location of, data used by the instruction. For example, an instruction that compares two characters to see if they are the same must specify which characters are to be compared. The general subject of specifying where the data is (i.e., its address) is called addressing and is discussed in Section 3.3.

Figure 3-14 shows several typical formats for level 2 instructions. On some level 2 machines, all instructions have the same length; on others, there may be two or three lengths. Moreover, instructions may be shorter, the same length as, or longer than the word length. Some possible relations between instruction length and word length are shown in Fig. 3-15.

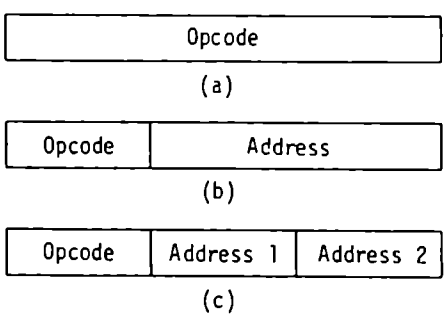


Fig. 3-14. Three typical instruction formats: (a) 0 address instruction; (b) 1 address instruction; (c) 2 address instruction.

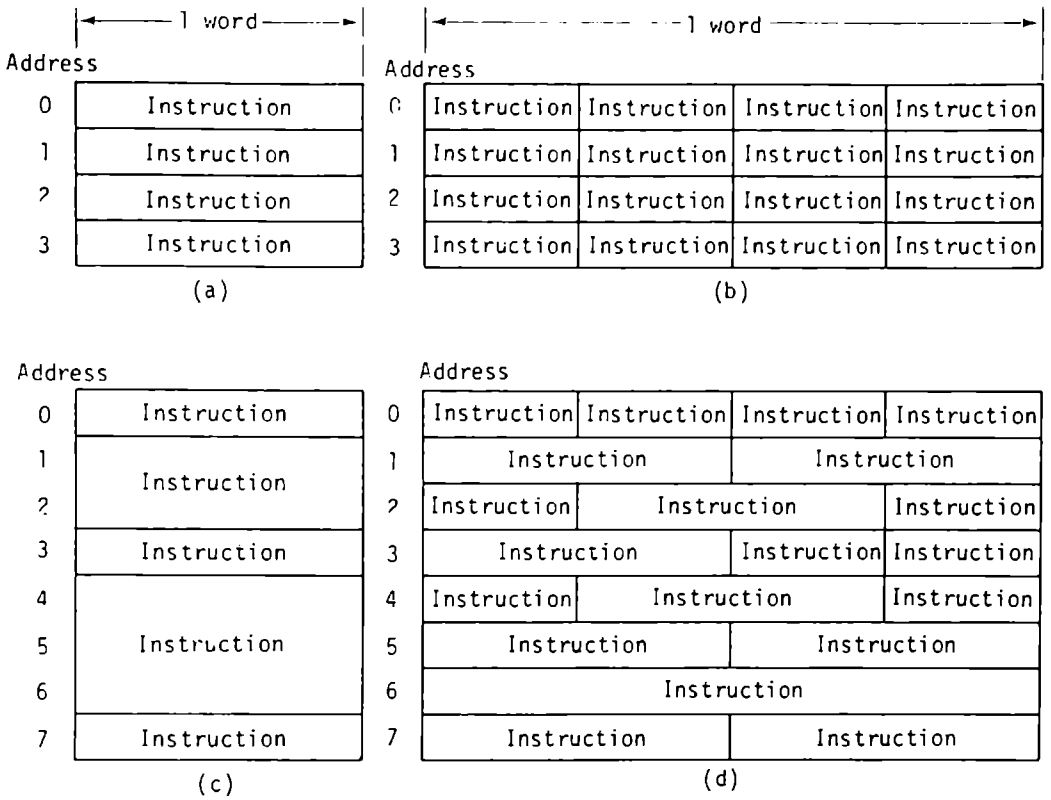


Fig. 3-15. Some possible relationships between instruction length and word length.

3.2.1. Design Criteria for Instruction Formats

When a computer design group has to choose an instruction format (or formats) for their machine, a number of factors must be considered. First, and most important, short instructions are better than long instructions. A program consisting of N 16-bit instructions takes up only half as much memory space as N 32-bit instructions. Memory is not free after all, so designers do not like to waste it unnecessarily.

A second reason is that each memory has a particular transfer rate that is determined by the technology and engineering design of the memory. The transfer rate of a memory is the number of bits per second that can be read out of the memory. A fast memory can give a processor (or an i/o device) more bits per second than a slow memory.

If the transfer rate of a particular memory is T bits per second, and the average instruction length is L bits, then the memory can deliver T/L instructions per second at most. This means that the rate at which instructions are executed (i.e., the processor speed) depends on the instruction length. Shorter instructions mean a faster processor. If the time required to execute an instruction is very long compared to the time required to fetch it from memory, then the instruction fetch time will not be so

important. With present-day memories and processors, the reverse is sometimes true: the processor is faster than the memory; consequently, increasing the number of instructions fetched per second is an important design criterion.

Sufficient room in the instruction to express all the operations desired is a second design criterion for instruction formats. A machine with 2^N operations and an instruction smaller than N bits is impossible. There simply will not be enough room in the opcode.

Still a third criterion is that it is highly desirable for the machine's word length to be an integral multiple of its character length. If the character code has K bits, the word length should be K or 2K or 3K or 4K or 5K and so on; otherwise space will be wasted when characters are stored. Of course, it is possible to store 3.5 characters per word, but doing so causes severe inefficiencies in accessing the characters. The restrictions placed on the word length by the character code affect the instruction length, since either an instruction should occupy an integral number of bytes or words, or an integral number of instructions should fit in a word. A design with a 7-bit character, a 12-bit instruction, and a 31-bit word would probably be a catastrophe.

A fourth criterion concerns the number of bits in an address field. Consider the design of a machine with an 8-bit character (possibly seven bits plus parity) and a main memory that must hold 2^{16} characters. The designers could choose to assign consecutive addresses to units of 8, 16, 24, or 32 bits, as well as other possibilities.

Imagine what would happen if the design team degenerated into two warring factions, one advocating making the 8-bit byte the basic unit of memory, and the other advocating the 32-bit word as the basic unit of memory. The former group would propose a memory of 2^{16} bytes, numbered 0, 1, 2, 3, . . . , 65,535. The latter group would propose a memory of 2^{14} words, numbered 0, 1, 2, 3, . . . , 16,383. Both memory organizations have the same bit capacity.

The first group would point out that in order to compare two characters in the 32-bit word organization, the program would not only have to fetch the words containing the characters but would also have to extract each character from its word so as to compare them. Doing so costs extra instructions and therefore wastes space. The 8-bit organization, on the other hand, provides an address for every character, thus making the comparison much easier.

The 32-bit word supporters would retaliate by pointing out that their proposal requires only 2^{14} separate addresses, giving an address length of only 14 bits, whereas the 8-bit byte proposal requires 16 bits to address the same memory. A shorter address means a shorter instruction that not only takes up less space but also requires less time to fetch. Alternately, they could retain the 16-bit address to reference a memory four times as large as the 8-bit organization allows.

This example demonstrates that in order to gain a finer memory resolution, one must pay the price of longer addresses, which, in general, means longer instructions. The ultimate in resolution is a memory organization in which every bit is directly addressable, (e.g., the Burroughs B1700), and at the other extreme is a memory consisting of very long words (e.g., the Cyber 70).

3.2.2. Expanding Opcodes

In the preceding section we saw how short addresses and good memory resolution could be traded off against each other. In this section we will examine tradeoffs involving both opcodes and addresses. Consider an N + K bit instruction with a K-bit opcode and a single N-bit address. This instruction allows 2^K different operations and 2^N addressable memory cells. Alternately, the same N + K bits could be broken up into a K − 1 bit opcode and an N + 1 bit address, meaning only half as many instructions but either twice as much memory addressable, or the same amount of memory but with twice the resolution. A K + 1 bit opcode and an N − 1 bit address give more operations; however, the price is either a smaller number of cells addressable or poorer resolution and the same number of cells addressable. Quite sophisticated tradeoffs are possible between opcode bits and address bits, in addition to the simpler ones just described. The scheme discussed in the following paragraphs is called an **expanding opcode**.

The concept of an expanding opcode can be most clearly seen through an example. Consider a machine in which instructions are 16 bits long and addresses 4 bits long, as shown in Fig. 3-16. This situation might be reasonable for a machine that had 16 registers (hence a 4-bit register address) on which all arithmetic operations took place. One design would be a 4-bit opcode and three addresses in each instruction, giving 16 three-address instructions. However, if the designers need 15 three-address instructions, 14 two-address instructions, 31 one-address instructions, and 16 instructions with no address at all, then they can use opcodes 0 to 14 as three-address instructions but interpret opcode 15 differently (see Fig. 3-17).

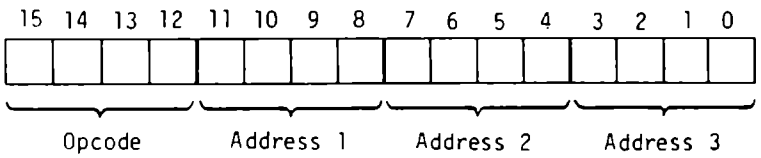


Fig. 3-16. An instruction with a 4-bit opcode and three address fields of four bits each.

Opcode 15 means that the opcode is contained in bits 8 to 15 instead of 12 to 15. Bits 0 to 3 and 4 to 7 form two addresses, as usual. The 14 two-address instructions all have 1111 in the leftmost four bits, and numbers from 0000 to 1101 in bits 8 to 11. Instructions with 1111 in the leftmost four bits and either 1110 or 1111 in bits 8 to 11 will be treated specially. They will be treated as though their opcodes were in bits 4 to 15. The result allows 32 new opcodes. Since only 31 are needed, 1111 1111 1111 is interpreted to mean that the real opcode is in bits 0 to 15, giving 16 instructions with no address. Notice that as we proceeded through this discussion, the opcode kept getting longer and longer; that is, the three-address instructions have a 4-bit opcode, the two-address instructions have an 8-bit opcode, the one-address instructions have a 12-bit opcode, and the zero-address instructions have a 16-bit opcode. Figure 3-13 illustrates how the PDP-11's expanding opcodes are organized.

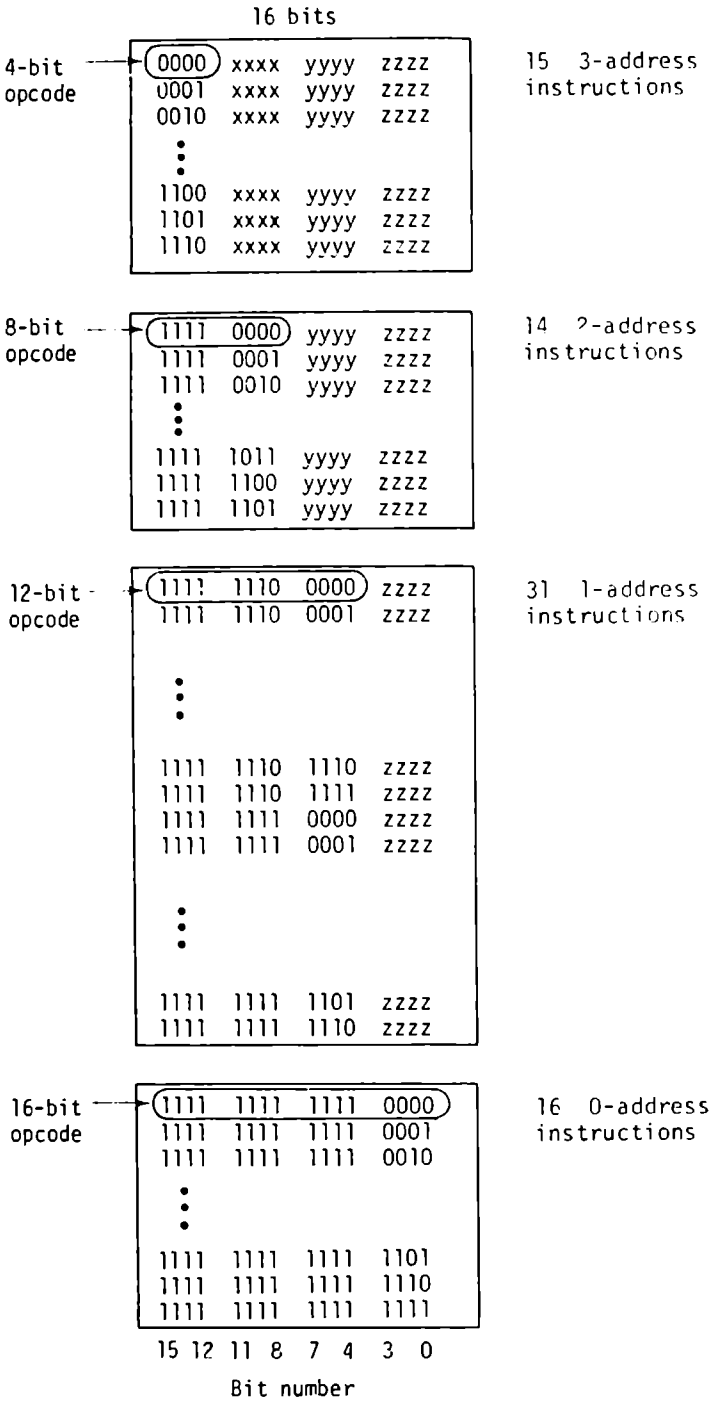


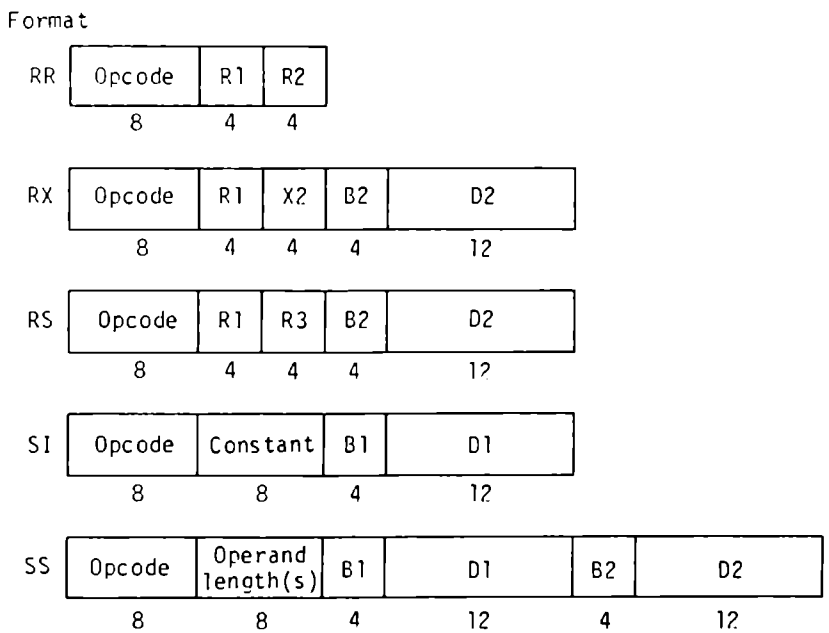
Fig. 3-17. An expanding opcode allowing 15 3-address instructions, 14 2-address instructions, 31 1-address instructions, and 16 0-address instructions. The fields marked xxxx, yyyy, and zzzz are 4-bit address fields.

Computer designers frequently leave room in the instruction set for new instructions to be added in the future. In a well-designed computer, an instruction with a nonexistent opcode should be treated as an error and some appropriate action taken.

A variation on the expanding opcode idea is to have some part of the opcode specify the instruction format and length. For example, the two leftmost bits might indicate instruction lengths of 8, 16, 32, or 64 bits, depending on which of the four values 00, 01, 10, and 11 they contained. In this case, there is no tradeoff between opcode and address. Instructions that need a long address part simply take up more space than instructions that need a short address part.

3.2.3. Examples of Instruction Formats

In this section we will examine the instruction formats of the IBM 370, Cyber 70, and PDP-11 conventional machine levels. The 370 level 2 machine has five major instruction formats (Fig. 3-18), as well as one other format that it is not important for our purposes (S format). The RR format is 16 bits, the RS, RX, and SI formats are 32 bits, and the SS format is 48 bits. In this context, R stands for register, S for



Operand length(s) may contain 2 4-bit lengths or 1 8-bit length;
R1, R2, R3 specify registers for operands and results;
B1, B2 specify base registers used in addressing;
D1, D2 are positive displacements from 0 to 4095 used for addressing memory in conjunction with a base register

Fig. 3-18. Principal instruction formats used on the 370. The numbers indicate the lengths of each field.

storage, X for index, and I for immediate. The length of an instruction is related to the number of memory addresses it uses. The basic instruction length is 16 bits, with an additional 16 bits for each memory address specified. The opcode is exactly eight bits long for all these instructions, with the leftmost two bits specifying the instruction length and formats as follows:

00 = RR format (16-bit instructions)
01 = RX format (32-bit instructions)
10 = RS or SI format (32-bit instructions)
11 = SS format (48-bit instructions)

The six remaining opcode bits are used to distinguish among the various instructions of each type. Six bits means a maximum of 64 instructions in each of the four classes listed above.

The second byte of the 370 instructions consists either of two 4-bit fields specifying registers or operand lengths, or one 8-bit field specifying a constant or an operand length. The second and third half words always contain the specification of a memory address, using a 4-bit field for a base register and a 12-bit field for a displacement.

The Cyber 70 has five instruction formats, as illustrated in Fig. 3-19. CDC does not provide any special names for them. Most instructions use the first format with three registers. This format has only six opcode bits, allowing for a maximum of 64

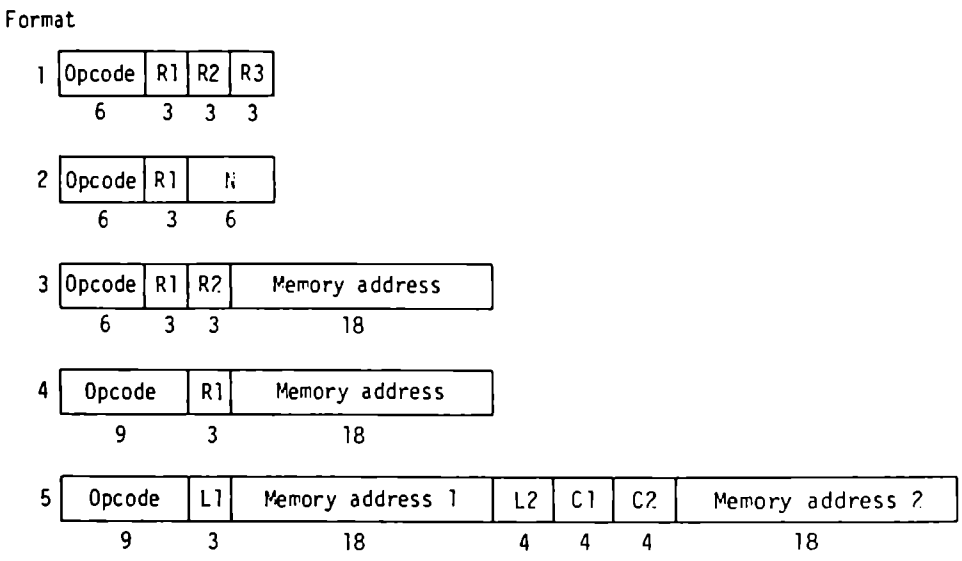


Fig. 3-19. Instruction formats used on the Cyber 70. The numbers below the fields indicate their lengths.

instructions. The second format is very similar except that in it the rightmost six bits are interpreted as one 6-bit number instead of two 3-bit numbers. The next two formats are used for instructions that reference memory. The last format, which is used by four highly specialized character move instructions, is one of the principal differences between the 6000 and Cyber 70 series computers (the 6000 series lacks it).

Despite the fact that the PDP-11 is a minicomputer, its instruction formats and addressing techniques are highly sophisticated. Figure 3-20 shows the major PDP-11 instruction formats in a way comparable to those of the IBM 370 and Cyber 70, although, from a narrow technical perspective, all instructions are actually 16 bits, and what appear to be 32- or 48-bit instructions are really 16-bit instructions followed by one or two data words. This point will be discussed in detail in Section 3.3.8.

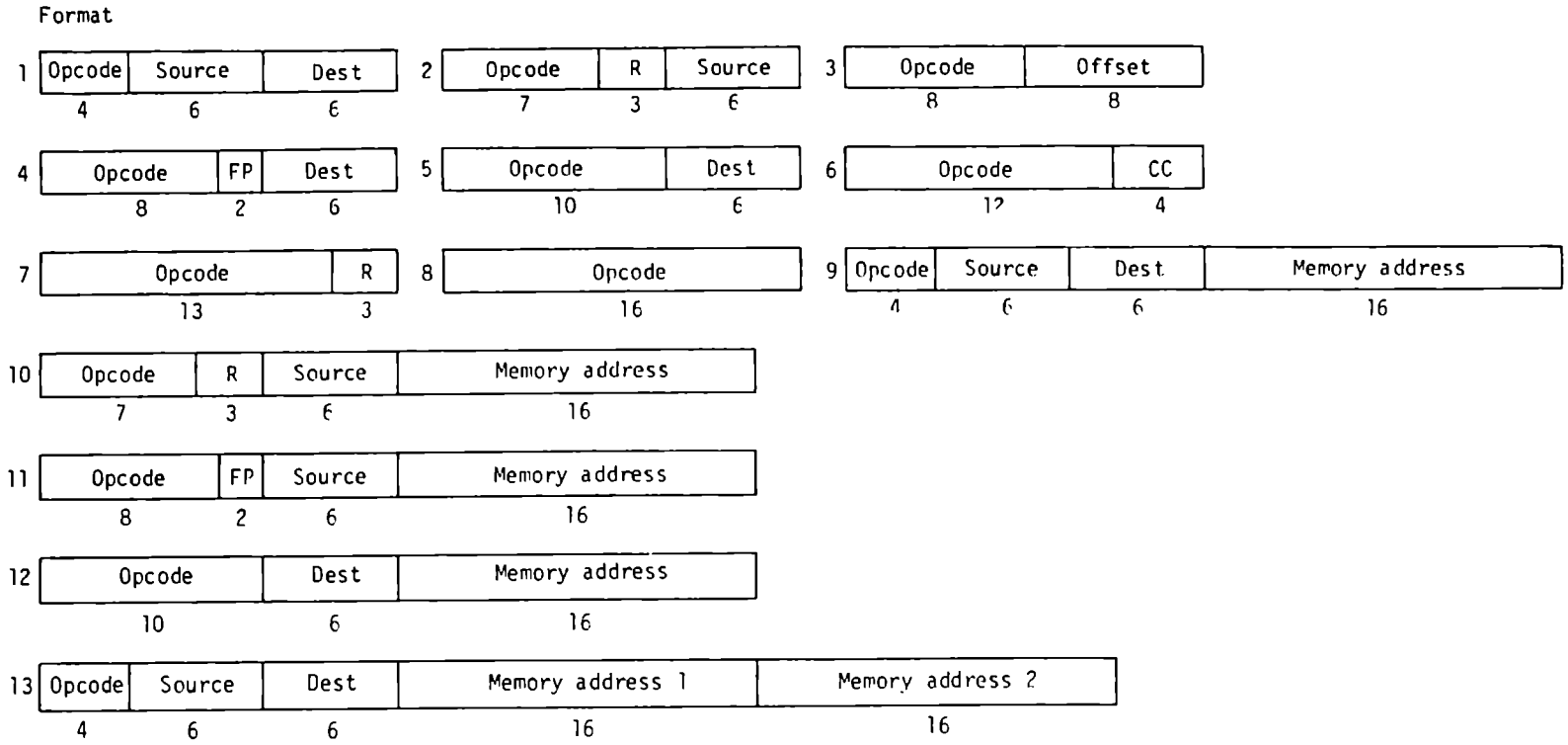
The PDP-11 uses an expanding opcode scheme. If bits 12 to 14 contain 1 to 6, the instruction is a double operand instruction; otherwise the leftmost 7, 8, or 10 bits are used for the opcode. The fields marked "source" and "dest" in Fig. 3-20 each contain two 3-bit fields, one specifying the addressing mode (to be discussed in Section 3.3.8) and the other specifying one of the eight general registers.

IBM's decision to have an eight-bit opcode was strongly suggested by the byte orientation of the entire addressing mechanism. Not all the 256 possibilities are used, however; but more than 128 are used, so a seven-bit opcode would not be adequate. The decision to use the first two bits of the opcode to distinguish the RR, RX, RS, SI, and SS formats limited each of those classes to 64 instructions, which was again just adequate. If three bits had been used, RS and SI could have been separated more easily, but 32 instructions would not have been adequate for the RR or RX instructions unless the floating-point instructions were put in a different group from the fixed-point instructions.

The Cyber 70, unlike the IBM 370, does not have an opcode that contains more than enough bits to encode all the instructions. The Cyber 70 has 78 instructions, which obviously is too many for six bits. A seven-bit opcode would be awkward, because most instructions require three registers, each of which requires three bits to specify. Unlike the IBM 370, the Cyber 70's registers are not general purpose, and the instruction that adds two X registers cannot add two B registers. Therefore specifying a register only requires three bits because the type of register to be used is determined by the opcode. A 7-bit opcode plus three 3-bit register fields yields a 16-bit instruction, which does not fit neatly in a 60-bit word the way a 15-bit instruction does.

In the original CDC 6600, the ancestor of all the 6000, 7000, Cyber 70, and Cyber 170 computers, there were fewer instructions. If one counted all eight conditional jump instructions as a single instruction, there were exactly 64 instructions, so a six-bit opcode was adequate. In order to distinguish between the eight conditional jump instructions, format 4 in Fig. 3-19 is used instead of format 3; that is, instead of indicating a register, the leftmost three-bit field became part of the opcode and selected one of the eight conditional jump instructions.

This slight expansion of the opcode is not without its price, however. If another three-bit field had been available, the conditional jump instructions could have used



Source and dest each contain a 3-bit addressing mode field and a 3-bit register number;
FP is 1 of the floating point registers 0, 1, 2, or 3;
R is 1 of the general registers;
CC is the condition code field

Fig. 3-20. Instruction formats used on the PDP-11. The numbers below the fields indicate their lengths.

it in the same way the unconditional jump instruction used it, thereby providing a more general and symmetric instruction set. The designers of the 6600 were, of course, aware of this fact, but there were simply no bits left over.

When CDC decided to add four character manipulation instructions to the series, it became necessary to find an available opcode for each one, in spite of the fact that every combination of six bits was already in use as an opcode. On the original machines, the NO OPERATION instruction used format 1 in Fig. 3-19, with opcode 46 (octal) and the R1, R2, and R3 fields ignored. All the instructions from 46000 to 46777 (octal) were treated as NO OPERATION (which is used primarily to fill out the last 15 bits of a word when the next instruction is 30 bits). In order to fit the four character manipulation instructions in somewhere, it was necessary to redefine the NO OPERATION instruction to be only 46000 (octal) so that 464xx, 465xx, 466xx, and 467xx (octal) could be used for the character manipulation instructions.

This redefinition invalidated existing 6000 series programs that used any of the alternative forms of NO OPERATION. In other words, the lack of encoding space ruined the possibility of having the Cyber 70s being 100 percent upward compatible with the 6000 series. Because no opcodes were available, programs exist that run correctly on the 6000 series but that do not run correctly on the Cyber 70s. It should be pointed out, however, that the alternative forms of NO OPERATION were almost never used.

Many years after the first 6600 had been sold, CDC decided that an integer multiply instruction might be a useful item to have. (One suspects that the lack of an available opcode may have influenced its absence in the first place.) Because the arithmetic instructions require three register fields (two for operands, one for results), the remaining 46000 opcodes were too short (by one bit) and so CDC settled for sharing instruction 42xxx between floating-point multiply and integer multiply. Depending on the numbers to be multiplied, the Cyber 70 chooses an integer multiplication or a floating-point multiplication. Consequently, the meaning of this opcode depends on the data.

The preceding examples have been discussed in order to illustrate the difficulties that can occur when not enough bits are available in which to encode all the instructions. By way of contrast, the IBM 370 has many unused opcodes that will serve IBM in good stead should the need arise to add new instructions in the future.

The problem of insufficient encoding room also occurs on the PDP-11 despite its large variety of instruction formats. Although there are six floating-point registers, instruction format 11 in Fig. 3-20 only provides two bits for the FP field, which is used to select a floating-point register. As a result, these instructions may only use floating-point registers 0 to 3 in this field.

3.3. ADDRESSING

Instructions can be classified according to the number of addresses they use. It should be kept in mind that a collection of numbered CPU registers, in effect, forms a high-speed memory and defines an address space. An instruction that adds register

1 to register 2 should be classified as having two addresses because the instruction must specify which registers are to be added, just as an instruction that adds two memory words must specify which words.

Instructions specifying 1, 2, and 3 addresses are all common. The three-address instructions specify the locations of two operands and the location where the result is to be put. The Cyber 70 format 1 instruction in Fig. 3-19 provides three registers, *i*, *j*, and *k*. The integer ADD instruction

```
IXi Xj + Xk
```

is described by the PL/I statement

```
REGISTER(i) = REGISTER(j) + REGISTER(k)
```

The 370 and PDP-11 both use two-address instructions for arithmetic and logical operations. The 370 instruction for adding registers *i* and *j*

```
AR i, j
```

means

```
REGISTER(i) = REGISTER(i) + REGISTER(j)
```

whereas the PDP-11 integer add instruction

```
ADD Ri, Rj
```

means

```
REGISTER(j) = REGISTER(i) + REGISTER(j)
```

Clearly, the 370 and PDP-11 designers had different taste. A three-address format is more flexible than a two-address format, but it also requires more bits.

On machines that do arithmetic with only one address, a special register called the **accumulator** provides one of the operands. On these machines, the address is usually the address of a memory word M, rather than a register. The instruction for addition specifying the address M has the effect

```
ACCUMULATOR = ACCUMULATOR + MEMORY(M)
```

Up to this point we have paid relatively little attention to how the bits of an address field are interpreted to find the operand. One possibility is that they contain the memory address of the operand. There are, however, other possibilities as well, and in the following sections we will explore some of them.

3.3.1. Immediate Addressing

The simplest way for an instruction to specify an operand is for the address part of the instruction actually to contain the operand itself rather than an address or other information describing where the operand is. This is called an **immediate operand** because the operand is automatically fetched from memory at the same time the instruction itself is fetched; hence it is immediately available for use.

Immediate addressing has the virtue of not requiring an extra memory reference to fetch the operand. It has the disadvantage of restricting the operand to a number

that can fit in an address field. In an instruction with a three-bit address (e.g., register field), the operands would be restricted to three bits, which rather limits their usefulness.

The IBM 370 conventional machine level has move, compare, and three boolean instructions (AND, OR, EXCLUSIVE OR) as well as several other instructions that contain a one-byte immediate operand.

The Cyber 70 has instructions to set an A, B, or X register from the sum of another register plus a constant. If the other register is B0, which always contains 0, the instructions become set register with a constant—that is, immediate instructions. There are three immediate instructions

SAi K, SBi K, and SXi K, (i = 0, 1, 2, 3, 4, 5, 6, or 7)

one for each type of register that can be loaded with a constant. The constant K is only 18 bits (see instruction format 3, Fig. 3-19), which is fine for the A and B registers but too short for the X registers. The instruction

SXi K

loads K into the lower 18 bits of Xi and copies the sign of K into the remaining 42 bits of Xi. The load register instructions are the only immediate instructions available on the Cyber 70.

In contrast, the PDP-11 can have an immediate operand not only for load register instructions but for store, boolean, compare, and all the arithmetic instructions as well. These instructions use formats 9, 10, and 13 in Fig. 3-20, with a 16-bit 2's complement constant in place of one of the addresses.

3.3.2. Direct Addressing

Another simple method for specifying an operand is to give the address of the memory word where the operand is contained. This process is called **direct addressing**. The details of how the computer knows which addresses are immediate and which are direct will be discussed later. On some machines, the opcodes for ADD IMMEDIATE and ADD DIRECT are different.

The 370 does not use direct addressing.

The Cyber 70 has only one instruction that uses direct addressing:

SAi K

which loads register Ai with the immediate operand ADDRESS. For i = 1, 2, 3, 4, or 5, the contents of word K is loaded into register Xi—that is, direct addressing. This is a somewhat unusual situation where one instruction has a single address that is simultaneously an immediate and a direct address. The constant K is loaded into the A register, making the instruction "load A register immediate"; and the contents of memory word K is fetched into the corresponding X register, qualifying the instruction to be considered "load X register direct."

The PDP-11 not only has direct address instructions with one address, but it also has a series of instructions that have two direct addresses. These instructions use

format 13 in Fig. 3-20 and include instructions for adding one memory word to another, comparing two memory words, and so on.

3.3.3. Register Addressing

Register addressing is conceptually the same as direct addressing. In this form of addressing, the address field contains the number of the register in which the operand is stored. A machine with 16 registers and 65,536 memory words really has two address spaces. One might think of an address on such a machine as having two parts: (a) one bit telling whether a register or a memory word is desired and (b) an address field telling which register or memory word is desired. Since there are fewer registers than memory words, a smaller address is needed, and thus a different format instruction is often used for register operands and memory operands.

If there were a corresponding register instruction for every instruction that addressed memory, half the opcodes would be for memory operands and half for register operands. One bit would be needed in the opcode to designate which address space was to be used. If this bit were then removed from the opcode field and placed in the address field, the fact that two address spaces were being used would be clearer.

Machines are designed with registers for two reasons: (a) registers are faster than main memory and (b) because there are so few of them, only a few bits are needed to address them. Unfortunately, having 8 or 16 registers also greatly complicates programming because decisions must be made as to which operands and which intermediate results are to be kept in the limited number of registers and which are to be kept to main memory. W. L. van der Poel (1968) has astutely remarked that computers should be provided with either 0, 1 or enough of each feature that programmers need not waste time thinking about what to do if they run out (the so-called 0, 1, or infinity rule).

The 370, Cyber 70, and PDP-11 all have a wide repertoire of instructions that take their operands from registers and leave their results in a register. A typical instruction is ADD. These instructions are illustrated in Fig. 3-18 as RR, Fig. 3-19 as format 1, and Fig. 3-20 as format 1. In addition, the IBM 370 and PDP-11 have register-memory instructions in which one operand is in a register.

3.3.4. Indirect Addressing

Direct addressing is a scheme in which the address specifies which memory word or register contains the operand. Indirect addressing is a scheme in which the address specifies which memory word or register contains not the operand but the address of the operand. As an example, consider the PDP-11 instruction to load register R1 indirectly from memory location 1000, where location 1000 contains 1510, as shown in Fig. 3-21(a).

First, the contents of location 1000 is fetched into an internal CPU register. This 16-bit number (1510) is not put in R1 (if it were, we would have a direct address instruction). Instead the contents of location 1510 is fetched and put into R1. The contents of location 1000 is not the operand, but instead it "points to" the operand.

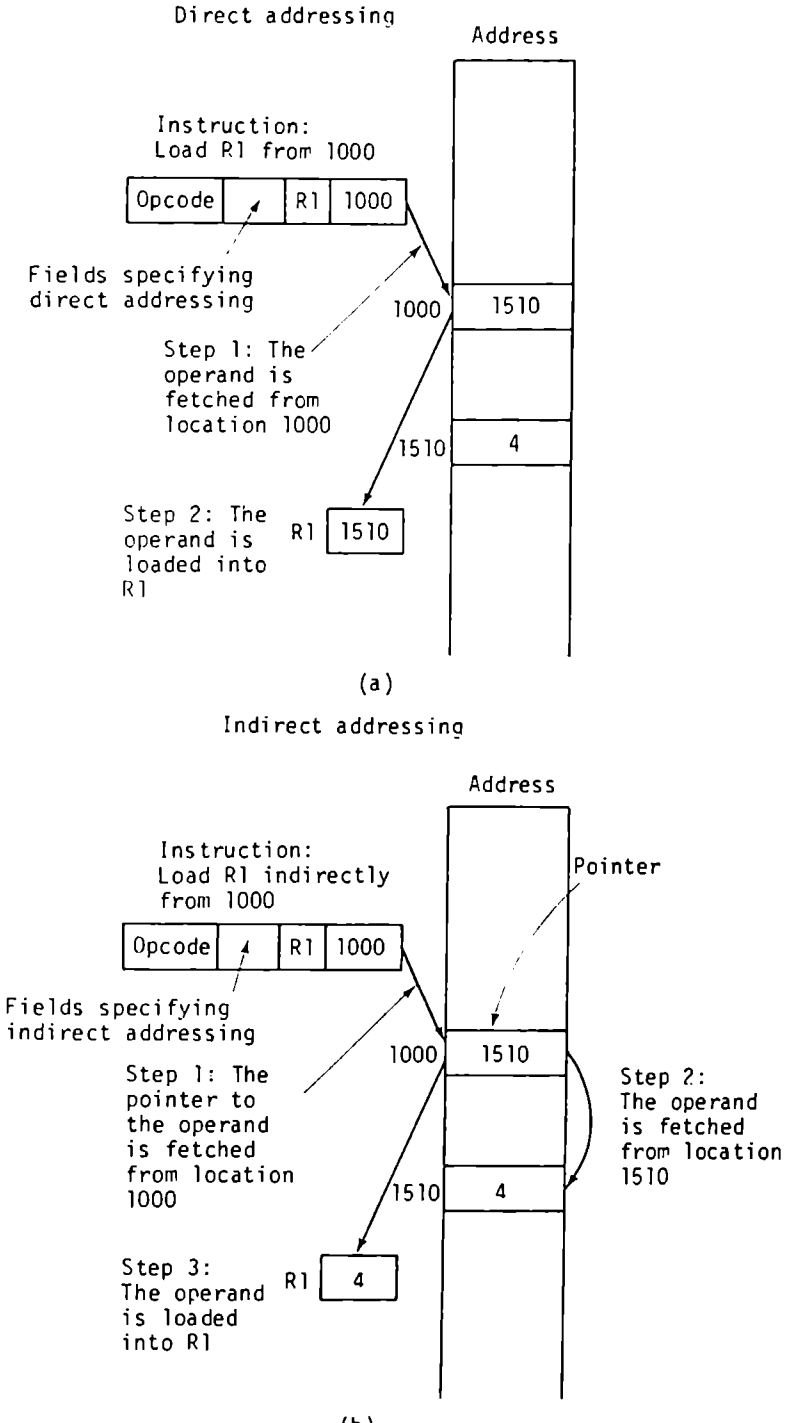


Fig. 3-21. Comparison of direct and indirect addressing. (a) Direct addressing. (b) Indirect addressing. The instruction format is the PDP-11's.

For this reason, it is called a **pointer**. The 370 and Cyber 70 do not have indirect addressing.

Some machines, although none of our three examples, have multiple-level indirect addressing. If the word containing the actual address also has the indirect bit on, the address it contains is used as a pointer to still another word. This process goes on indefinitely, until a word without the indirect bit is found.

There is a certain progression between immediate, direct, indirect, and multiple-level indirect addressing. Immediate addressing requires zero memory references, since the operand was fetched along with the instruction, which we will not count here. Direct addressing requires one memory reference, to fetch the operand. Indirect addressing requires two memory references, one for the pointer and one for the operand. Multiple-level indirect addressing requires at least three memory references, two or more for pointers and one for the operand. Memory references in this context include register references.

3.3.5. Indexing

Many problems require performing some operation on a sequence of data structures stored in consecutive memory locations. For example, consider a block of N machine words occupying locations A, A + 1, A + 2, . . . , A + N − 1, which must be moved to locations B, B − 1, B + 2, . . . , B + N − 1. Assuming the machine has an instruction

```
MOVE A,B
```

which moves the contents of location A to location B, one could execute this instruction, modify the instruction itself to read

```
MOVE A + 1,B + 1
```

execute it again, then modify it again and repeat the cycle until all N words had been copied.

Although programs that modified themselves were once popular, they are now considered bad programming practice. They are difficult to debug, and they make sharing of a program among several users of a time-sharing system difficult, among other reasons.

The copying problem could also be solved by indirect addressing. One register or memory word would be loaded with the address of A; a second one would be loaded with the address of B. The MOVE instructions would use these two as pointers. After each word was copied, the pointers would each be increased by one. The pointers are part of the data, not part of the program, of course.

Another solution is to have one or more registers called **index registers**, which work as follows. Addresses have two parts: the number of an index register and a constant. The address of the operand is the sum of the constant and the contents of the index register. In the example above, if both addresses were indexed using an index register containing the integer K, the instruction MOVE A, B would move the contents of memory location A + K to B + K. By initializing the index register to

0 and incrementing it by 1 after each word is copied, only one register is needed for the copy loop. Furthermore, incrementing a register is faster than incrementing a memory location.

The IBM 370 RX instructions can use any of the general registers except 0 as an index register. The instructions that perform arithmetic or boolean operations using a register and a memory location may have indexing specified for the memory location. The index register to be used is specified in the X2 field of the RX instruction. If the X2 field contains a 0, no indexing is performed, analogous to the use of B0 on the Cyber 70. After the address and index register are added, only the low-order 24 bits are used as the address.

The Cyber 70 has seven nonzero B registers, which are really index registers. The instruction

```
SAi Bj + K
```

adds the contents of B*j* and the constant K and puts the result in A*i*. If *i* is 1, 2, 3, 4, or 5, a fetch occurs; if *i* is 6 or 7, a store occurs. This amounts to an indexed load or store of an X register.

All the PDP-11 instructions that contain a memory address can have that address indexed, using any of the eight general registers. Moreover, instructions with two memory addresses may specify a different index register for each address, unlike the IBM 370 SS format instructions, which may not be indexed at all.

In the preceding copying example it was necessary to explicitly increment the index register by 1 after each use of it. The need to increment or decrement an index register just before or after it is used is so common that some computers provide special instructions, or addressing modes, or even special index registers, which automatically increment or decrement themselves. Automatic modification of an index register is called **autoindexing**. The PDP-11 can use any of its general registers for autoindexing. This topic will be discussed in more detail in Section 3.3.8.

3.3.6. Base Registers

The distribution of memory references made by most programs is highly nonuniform. Most fetches and stores come from a few data areas. Explicit references to the program itself (except for jumps) are rare. The 370 and some other machines exploit this nonuniformity in order to reduce the number of bits in the addresses used at the conventional machine level.

The address length can be reduced by having a number of **base registers**, each containing a pointer to some memory location. Addresses consist of two parts: a base register number and an integer called the **displacement**. The addressing computation adds the displacement to the contents of the selected base register to produce the address of the operand. A base register must contain sufficiently many bits to allow it to point to any memory address. If there are N base registers, and the displacement contains K bits, the programmer will be able to choose any N regions of 2^K addresses each to be accessible at any one instant.

Figure 3-22 illustrates this situation for the case of four base registers and a 12-bit displacement. The address in this example is actually 14 bits—2 bits to specify a base register and 12 bits to specify a displacement. A 14-bit address can specify 2^{14} memory locations. In Fig. 3-22, four regions of 2^{12} addresses each are accessible, making 2^{14} in all. The shaded areas can be accessed, but the areas in between are impossible to access. If the program needs data from one of them, it must first change a base register to point to some address not more than 4096 addresses before the desired location.

The astute reader may have noticed that in both indexing and base register addressing, the contents of the selected register is added to a constant contained in the instruction itself in order to produce the address of the operand. Thus there appears to be no difference between the two modes of addressing. This is, in fact, true. Traditional usage is that when the displacement has enough bits to address the entire

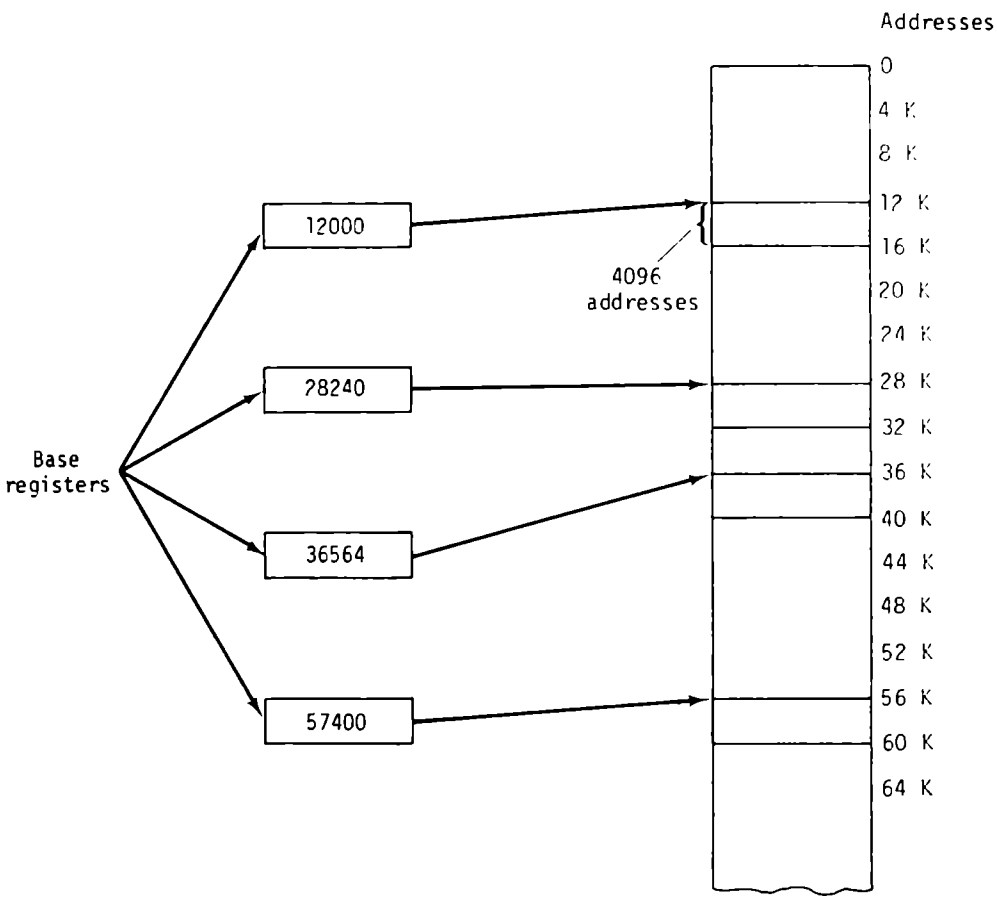


Fig. 3-22. Base register addressing using four base registers and a 12-bit positive displacement. The four regions of 4K that are accessible without changing a base register are shaded.

memory itself, the scheme is called indexing. Otherwise it is called base register addressing. Furthermore, base register addressing schemes usually allow two registers to be specified, the operand address being the sum of both registers and the displacement. One of the registers can then be used for indexing through memory as in the example of copying a block of words from A to B.

The IBM 370 computers use base register addressing, with 16 registers and a 12-bit displacement. The addressable unit is the byte, so a maximum of 2^{16} bytes may be addressable at one time. Actually, fewer are addressable, because the general registers are also used for index registers, procedure calls, and other purposes. Consequently, not all 16 are available for use as base registers. The total address space consists of 2^{24} bytes; therefore less than one percent of the address space is available at any one instant. However, if the program avoids spreading its data over too wide a range of addresses, this percent may be more than adequate. A 4-bit base register field and a 12-bit displacement field save eight bits compared to a 24-bit direct address.

3.3.7. Stack Addressing

We have already noted that having machine instructions be as short as possible is highly desirable for saving both CPU time and memory. The ultimate limit in reducing address lengths would be to have instructions with no addresses at all, just opcodes. Surprisingly enough, this situation is possible. It is accomplished by organizing the machine around a data structure called a stack.

A **stack** consists of data items (words, characters, bits, etc.) stored in consecutive order in the memory. The first item pushed onto the stack is said to be at the bottom of the stack. The item most recently pushed onto the stack is said to be on the top of the stack. There is always a register or a memory word that contains the address of the top of the stack. It is called the **stack pointer.**

Figure 3-23 illustrates the operation of a stack. In Fig. 3-23(a), there are already two items on the stack. The bottom of the stack is at memory location 1000, and the top of the stack is at memory location 1001. The stack pointer contains the address of the item on the top of the stack, namely, 1001; that is, it "points" to the top of the stack. In Fig. 3-23(b), 6 has been pushed onto the stack, and the stack pointer indicates 1002 as the new top of the stack. In Fig. 3-23(c), 75 has been pushed onto the stack, raising the stack pointer to 1003. In Fig. 3-23(d), 75 has been popped off the stack.

Computers that are stack oriented have an instruction to push the contents of a memory location or a register onto the stack. Such an instruction must both copy the item and increment the stack pointer. Similarly, an instruction to pop the top of the stack into a register or memory location must make a new copy in the proper place and decrement the stack pointer. Some computers have their stacks upside down, with new items being pushed into consecutively lower-numbered memory locations rather than consecutively higher-numbered locations, as in Fig. 3-23.

Instructions with no addresses are used in conjuction with a stack. This form of addressing specifies that the two operands are to be popped off the stack, one after

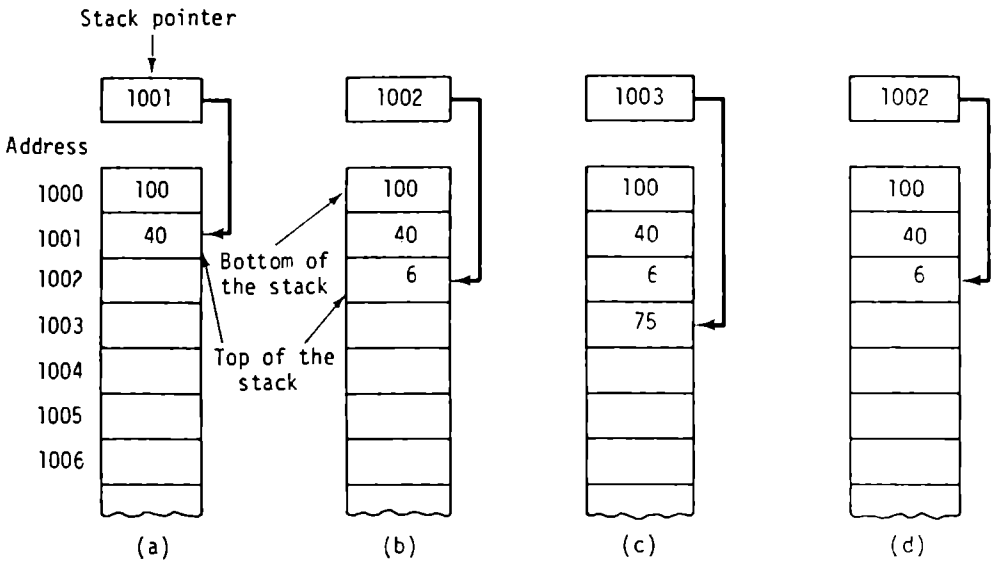


Fig. 3-23. Operation of a stack.

another, the operation performed (for example, multiplication or EXCLUSIVE OR) and the result pushed back onto the stack. Figure 3-24(a) shows a stack containing four items. A multiply instruction has the effect of popping the 5 and the 6 off the stack, temporarily resetting the stack pointer 1001, and then pushing the result, 30, onto the stack, as shown in Fig. 3-24(b). If an addition is then performed, the result will be as shown in Fig. 3-24(c).

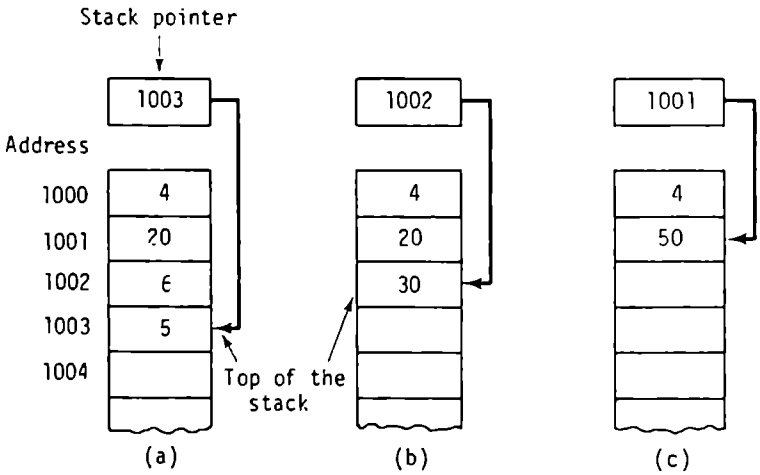


Fig. 3-24. Using a stack for arithmetic. (a) Initial configuration. (b) After a multiplication. (c) After an addition.

Reverse Polish

It is a long-standing tradition in mathematics to write the operator between the operands, as in X + Y, rather than after the operands, as in XY+. The form with the operator *in* between the operands is called **infix** notation. The form with the operator *after* the operands is called **postfix**, or, more commonly, **reverse Polish**, after the Polish logician J. Lukasiewicz (1958), who investigated the properties of this notation.

Reverse Polish has a number of advantages over infix for expressing algebraic formulas. First, any formula can be expressed without parentheses. Second, it is very convenient for evaluating formulas on computers with stacks. Third, infix operators have precedence. For example, we know that $a \times b + c$ means $(a \times b) + c$ and not $a \times (b + c)$, because multiplication has been arbitrarily defined to have precedence over addition. Reverse Polish eliminates this nuisance.

Several algorithms for converting infix formulas into reverse Polish exist. The one given below is an adaptation of an idea due to E. W. Dijkstra. Assume that a formula is composed of the following symbols: variables, the dyadic operators + − * /, and left and right parentheses. To mark the end of a formula, we will insert the symbol $\perp$ after the last symbol and before the first symbol.

Figure 3-25 shows a railroad track from New York to California, with a spur in the middle that heads off toward Texas. Each symbol in the formula is represented by one railroad car. The train moves westward (to the left). When each car arrives at

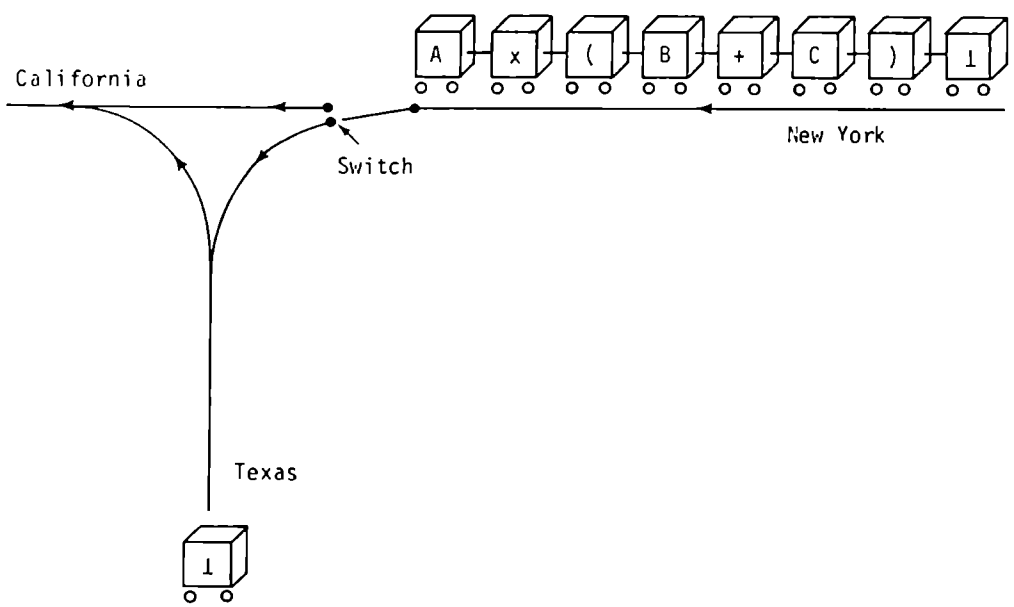


Fig. 3-25. Each railroad car represents one symbol in the formula to be converted from infix to reverse Polish.

Car at the switch

Most recently arrived car on the Texas line	⊥	+	−	x	/	(	)
⊥	4	1	1	1	1	1	5
+	2	2	2	1	1	1	2
−	2	2	2	1	1	1	2
x	2	2	2	2	2	1	2
/	2	2	2	2	2	1	2
(	5	1	1	1	1	1	3

Fig. 3-26. Decision table used by the infix to reverse Polish algorithm.

the switch, it must stop and ask if it should go to California directly or take a side trip to Texas. Cars containing variables always go directly to California and never to Texas. Cars containing other symbols must inquire about the contents of the nearest car on the Texas line before entering the switch. Figure 3-26 shows what happens, depending on the contents of the nearest car on the Texas line and the car at the switch. The first ⊥ always goes to Texas. The numbers refer to the following situations:

1. The car at the switch heads toward Texas.
2. The most recent car on the Texas line turns around and goes to California.
3. Both the car at the switch and the most recent car on the Texas line are hijacked and disappear; that is, both are deleted.
4. Stop. The symbols now in California represent the reverse Polish formula when read from left to right.
5. Stop. An error has occurred. The original formula was not correctly balanced.

After each action is taken, a new comparison is made between the car at the switch, which may be the same car as in the previous comparison or it may be the next car, and the car that is now the last one on the Texas line. The process continues until step 4 is reached. Notice that the Texas line is being used as a stack, with routing a car to Texas being a push operation, and turning a car already on the Texas line around and sending it to California being a pop operation.

Note that the order of the variables is the same in infix and reverse Polish. Figure 3-27 gives examples of infix formulas and their reverse Polish equivalents.

Evaluation of Reverse Polish Formulas

The following algorithm evaluates a reverse Polish formula.

ALGORITHM

1. Examine each symbol in the reverse Polish formula, starting at the extreme left until you come to an operator.

Infix	Reverse Polish
A + B x C	ABC x +
A x B + C	AB x C +
A x B + C x D	AB x CD x +
(A + B)/(C - D)	AB + CD - /
A x B/C	AB x C/
((A + B) x C + D)/(E + F + G)	AB + C x D + EF + G + /

Fig. 3-27. Examples of infix formulas and their reverse Polish equivalents.

2. Write down the operator and the two operands immediately to its left on a piece of scratch paper.
3. Erase the operator and the operands from the formula, creating a hole.
4. Perform the operation on the operands and write the result in the hole in the formula.
5. If the formula now consists of one value, that is the answer and the algorithm is finished; otherwise go to step 1.

Figure 3-28 depicts the evaluation of a reverse Polish formula. Note that the order of the operators is the order in which they are actually used.

Reverse Polish is the ideal notation for evaluating formulas on a computer with a stack. The formula consists of N symbols, each one either a variable (i.e., something with a value) or an operator. The algorithm is as follows.

Infix formula (8 + 2 x 5)/(1 + 3 x 2 - 4)

Reverse Polish formula 8 2 5 x + 1 3 2 x + 4 - /

Step	Formula to be evaluated	Leftmost operator	Left operand	Right operand	Result	New formula after performing operation
1	8 2 5 x + 1 3 2 x + 4 - /	x	2	5	10	8 10 + 1 3 2 x + 4 - /
2	8 10 + 1 3 2 x + 4 - /	+	8	10	18	18 1 3 2 x + 4 - /
3	18 1 3 2 x + 4 - /	x	3	2	6	18 1 6 + 4 - /
4	18 1 6 + 4 - /	+	1	6	7	18 7 4 - /
5	18 7 4 - /	-	7	4	3	18 3 /
6	18 3 /	/	18	3	6	6

Fig. 3-28. Evaluation of the infix formula $(8 + 2 \times 5)/(1 + 3 \times 2 - 4)$ by converting it to reverse Polish and then evaluating the reverse Polish formula.

ALGORITHM

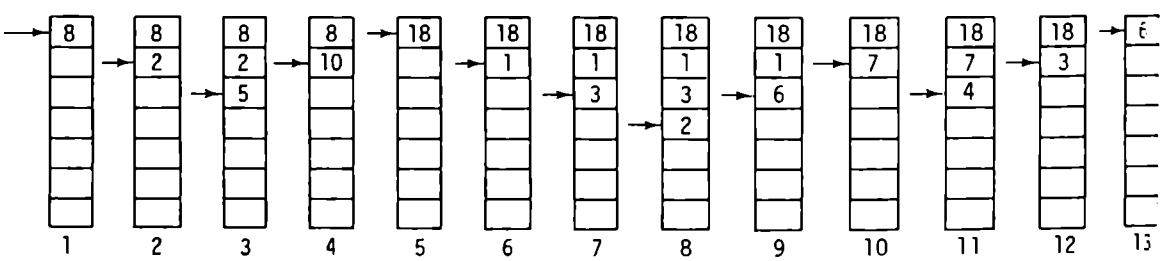
1. Set K to 1.
2. Examine the Kth symbol. If it is a variable, push it onto the stack. If it is an operator, pop the top two items off the stack, perform the operation, and push the result back onto the stack.

Infix formula (8 + 2 x 5)/(1 + 3 x 2 - 4)

Reverse Polish formula 8 2 5 x + 1 3 2 x + 4 - /

Step		
1	8 2 5 x + 1 3 2 x + 4 - /	Push 8
2	2 5 x + 1 3 2 x + 4 - /	Push 2
3	5 x + 1 3 2 x + 4 - /	Push 5
4	x + 1 3 2 x + 4 - /	Multiply 2 x 5
5	+ 1 3 2 x + 4 - /	Add 8 + 10
6	1 3 2 x + 4 - /	Push 1
7	3 2 x + 4 - /	Push 3
8	2 x + 4 - /	Push 2
9	x + 4 - /	Multiply 3 x 2
10	+ 4 - /	Add 1 + 6
11	4 - /	Push 4
12	- /	Subtract 7 - 4
13	/	Divide 18/3

(a)



(b)

Fig. 3-29. Use of a stack to evaluate a reverse Polish formula. (a) Steps in the evaluation process. (b) The stack after the corresponding step in (a). The arrow is the stack pointer.

3. If K = N, the algorithm terminates and the answer is on the stack. Otherwise add 1 to K and go to step 2.

Figure 3-29 shows the evaluation of the same formula as in Fig. 3-28 but using a stack this time. Note that the number on top of the stack is the right operand, not the left operand.

A computer organized around a stack offers several advantages compared to multiregister machines, such as the 370, Cyber 70, and PDP-11:

1. Short instructions, because many instructions have no addresses.
2. Formulas are easy to evaluate.
3. There is no need for complicated algorithms to optimize register use.

Although neither the 370, the Cyber 70, nor the PDP-11 have stack arithmetic instructions, some compilers for these machines use the registers to simulate a stack in software, at a certain cost in performance compared to having level 2 instructions to perform stack arithmetic directly.

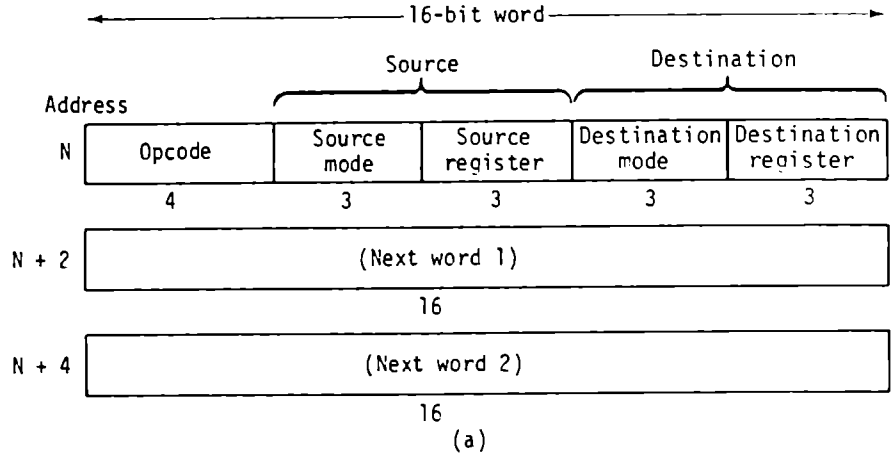
Although the PDP-11 does not have stack arithmetic instructions, it does have stack addressing. Register 6 is the main stack pointer, although other registers may be used as stack pointers for auxilliary stacks. The instructions to push a word onto a stack and pop a word off a stack use autoindexing on the stack pointer.

3.3.8. Addressing on the PDP-11

The previous sections discussed a number of different addressing modes—immediate addressing, direct addressing, indirect addressing, indexing, and so on. There still remains the question of how the hardware or level 1 interpreter knows whether an address is immediate, direct, indirect, indexed, and so on. One solution is to have a separate opcode for each addressing mode—that is, separate opcodes for ADD IMMEDIATE, ADD DIRECT, ADD INDIRECT, and so on. Another way is to make the mode part of the address. Each instruction could contain a few bits per address specifying which form of addressing was desired.

The PDP-11 has the largest number of addressing modes of the level 2 machines we have examined and is worth examining in detail to see how the addressing works. Figure 3-30(a) shows the format of the two-operand instructions, such as MOVE, ADD, COMPARE, and INCLUSIVE OR. Each operand has a three-bit addressing mode and a three-bit register field. The meaning of the modes is given in Fig. 3-30(b). Remember that register 7 is the program counter and that it is advanced by 2 immediately after an instruction word is fetched (before the instruction is executed).

All PDP-11 instructions are actually only 16 bits, but, in some cases, one or two extra words directly following the instruction are used by the instruction and can be considered part of it. Mode 6 and mode 7 addressing require a 16-bit constant for indexing. Furthermore, if either mode 2 or mode 3 is specified with register 7 (the program counter), the following sequence of steps occurs.



(a)

Mode	Name	How the operand is located
0	Register addressing	The operand is in R.
1	Register indirect	R contains a pointer to the operand.
2	Autoincrement	The content of R is fetched and used as a pointer to the operand. After this step, but before the instruction is executed, R is incremented by 1 (byte instructions) or 2 (word instructions).
3	Autoincrement indirect	The address of a memory word containing a pointer to the operand is fetched from R. Then R is incremented by 1 or 2 before the instruction is executed.
4	Autodecrement	R is first decremented by 1 or 2. The new value of R is then used as a pointer to the operand.
5	Autodecrement indirect	R is first decrementd by 1 or 2. The new value of R is then used as the address of a memory location containing a pointer to the operand.
6	Indexing	The operand is at the address equal to the sum of R (the index register) and the 16 bit 2's complement offset in the next word. In modes 6 and 7, the program counter (R7) is incremented by 2 immediately after the next word is fetched.
7	Indexing + indirect addressing	The memory location containing a pointer to the operand is found by adding the contents of R and the next word. In modes 6 and 7, the program counter (R7) is incremented by 2 immediately after the next word is fetched.

(b)

Fig. 3-30. (a) Format of a PDP-11 2-address instruction. (b) Description of the PDP-11 addressing modes. *R* is the register specified along with the mode.

First, the instruction is fetched and register 7 is increased by 2 (a word is two bytes). Then register 7 is used as a pointer to the data (mode 2) or the address of the data (mode 3). In both cases, the word pointed to is the word following the instruction. After this word is fetched, the autoincrementing of register 7 takes place, increasing register 7 by 2. Autoincrement addressing that specifies the program counter is a clever trick allowing the word following the instruction to be used as data. In mode 2 this data is the operand, yielding immediate addressing. In mode 3 this data is the address of the operand, yielding direct addressing. If both source and destination require an extra word, the first one is for the source.

The PDP-11 has an interesting form of addressing called **self-relative** or **position-independent** addressing. When mode 6 and register 7 are specified, the operand address is found by forming the sum of the index word following the instruction and the program counter. In effect, the index word gives the operand address by specifying how far away it is from the instruction itself, either in front or behind.

If all memory references use this form of addressing instead of direct addressing (mode 3 with register 7), a program can be loaded anywhere in memory and it will run correctly. In addition, it can also be moved after being loaded, because although the absolute addresses of the needed operands change, their distance from the instructions referencing them remains fixed.

As an example of the power of the PDP-11 addressing mechanism, consider the MOV instruction of Fig. 3-31(a). This instruction moves the source operand to register 4. Figure 3-31(b) shows all the different variations of this instruction for different source modes and registers.

Because both the source addressing mode and the destination addressing mode can be independently specified, a single opcode yields a large number of different instructions. For example, the ADD instruction can be used to

add a register to another register (0, 0)

add a register to a memory word (0, 6)

add a memory word to a register (6, 0)

add a memory word to another memory word (6,6)

pop a word from a stack and add it to a register (2, 0)

pop a word from a stack and add it to a memory word (2, 6)

add an immediate operand to a register (2, 0)

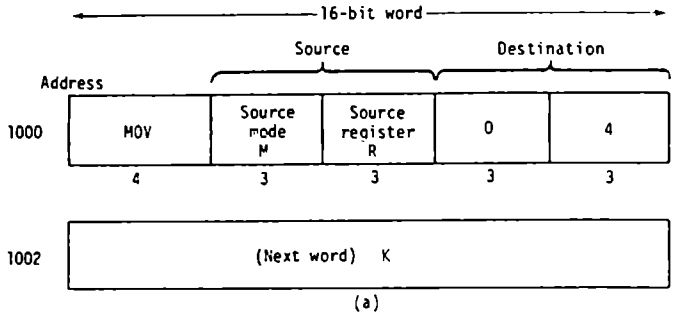
add an immediate operand to a memory word (2, 6)

add an immediate operand to the top word on the stack (2, 1)

add a register to the top word on the stack (0, 1)

add a memory word to the top word on the stack (6, 1)

add a memory word to an indirectly specified address (6, 7)



(a)

Source mode M	R = 0 - 5	R = 6	R = 7
	Source register		
0	Move R to R4 (register - register) Example: MOV R3, R4	Move stack pointer to R4 Example: MOV SP, R4	Move program counter to R4 Example: MOV PC, R4
1	Move memory word pointed to by R to R4 Example: MOV @R3, R4	Move top of the stack to R4, but do not remove it from the stack Example: MOV @SP, R4	Move K to R4; program counter is not incremented again so K will be executed as the next instruction Example: MOV @PC, R4
2	Move memory word pointed to by R to R4 and add 2 to R Example: MOV (R3)+, R4	Remove a word from the stack and put it in R4 (pop instruction) Example: MOV (SP)+, R4	Move K to R4 (immediate addressing) Example: MOV #24, R4
3	Move to R4 the memory word addressed by the word R points to, and add 2 to R Example: MOV @(R3)+, R4	Pop the address of the source operand from the stack, and move the source operand itself to R4 Example: MOV @(SP)+, R4	Load R4 from memory address K (direct addressing) Example: MOV @ #24, R2
4	Decrement R by 2 and then load R4 from the address R points to Example: MOV -(R3), R4	M = 4 and R = 6 is not useful as a source; however it is used as a destination in push instructions Example of push: MOV #6, -(SP)	Not used (causes an infinite loop)
5	Decrement R by 2 and then load R4 indirectly from the address R points to Example: MOV @-(R3), R4	Not used	Not used (causes an infinite loop)
6	Load R4 with the memory word at C(R) + K (indexing) Example: MOV 24(R3), R4	Load R4 with the word K/2 words below the top of the stack Example: MOV 24(SP), R4	Load R4 with the word K/2 words from this instruction (self relative addressing) Example: MOV X, R4 Note: The assembler computes the appropriate constant to address X
7	Load R4 with the memory word pointed to by C(R) + K (indexing + indirect addressing) Example: MOV @24(R3), R4	Load R4 from the word whose address is K/2 words below the top of the stack Example: MOV @24(SP), R4	Load R4 with the word pointed to by the word K bytes from this instruction (indirect addressing) Example: MOV @X, R4 Note: The assembler computes the appropriate constant to address X

(b)

Fig. 3-31. (a) A PDP-11 instruction that moves a word to register R4. (b) Different variations of this instruction for different source modes and registers. *R* is the register and *C(R)* is its contents. The assembly language notation is shown in the examples.

add a register to an indirectly specified address (0, 7)

add an immediate operand to an indirectly specified address (2, 7)

as well as numerous other possibilities. The numbers in parentheses are the source and destination modes. Note that mode 6 can always be replaced by mode 3; that is, memory can be addressed self-relative or directly. Also note that mode 1 with register 6 uses the top word of the stack as a source or destination but does not remove it from the stack, whereas mode 2 with register 6 does remove it.

Although only one opcode is used, the PDP-11 ADD instruction has dozens of distinct and useful variations. If all the variations of all 12 two-address instructions were counted as separate instructions, the PDP-11 would have hundreds of instructions. Because of the extreme flexibility of its addressing modes, the PDP-11 has a far more powerful and useful instruction set than the instruction list might indicate at first glance.

3.3.9. Discussion of Addressing Modes

In contrast to the PDP-11, the 370 and Cyber 70 (which were designed years before the PDP-11) lack indirect addressing, autoindexing, and stack addressing. Furthermore, the addressing modes of their operands are not independently specifiable. For example, the 370 has different opcodes for add memory to register and add register to register, and no instructions to add a register to a memory word or one memory word to another. (There is an instruction to add two numbers in memory in decimal format, however.)

The freedom of a computer designer to include or exclude certain types of addressing from a new machine is severely constrained by the number of bits available for encoding the addressing information. For instance, let us consider why the Cyber 70 does not have RX (register to memory, indexed) instructions, like the 370. These instructions require a memory address and two registers—one for the data and one for the indexing.

On the Cyber 70, a memory address uses 18 bits (although 17 would suffice) and a register 3 bits, making 24 bits for a memory address and two registers. A 30-bit instruction would leave only six bits over for the opcode. RX instructions for addition, subtraction, multiplication, or division of integers or floating-point numbers would each require a separate opcode, and as we have seen, spare opcodes are in short supply on the Cyber 70.

As another example, consider why the IBM 370 does not have three-register addressing as the Cyber 70 does. Adding a third four-bit register field to the RR instructions would have required extending those instructions to 20 bits, hardly a reasonable value considering that all the other instructions lengths are multiples of a half word. The RR instructions could have been extended to 32 bits, but doing so would have been very wasteful of space, unless a use could have been found for the remaining 12 bits.

Moreover, allowing three registers in the RR format (which would probably have

been renamed RRR format) would have produced a fundamental asymmetry between the three-address RRR instructions and the two-address RX instructions, thereby making programming more complicated. Of course, the RX instructions could have been made 48 bits and renamed RRX. By now it should be clear that adding a third register to the RR instructions would have required a major redesign of the 370.

As a third example, consider the possibility of adding eight more general registers to the PDP-11, so that it would have 16, just as the 370 does. Having 16 registers would have required a four-bit field to specify a register, instead of a three-bit field. The result would have increased the two-operand instructions to 18 bits and the one-operand instructions to 17 bits, rather awkward numbers considering the byte/word organization of the PDP-11.

The eight additional registers could have been accommodated by reducing the three-bit field specifying addressing mode to two bits, at the cost of eliminating several types of addressing. For example, the four allowed types could be register, indexing, autoincrement (which becomes immediate when the program counter is specified as the register), and autodecrement. Direct and indirect addressing would have been eliminated. Alternatively, they could have been included and the two autoindexing forms deleted, eliminating the stack addressing and immediate instructions. No matter what choice was made, the addressing power of the PDP-11 would have been severely weakened.

We have now completed our examination of the various tradeoffs possible between opcodes and addresses, and various forms of addressing. When approaching a new computer, you should examine the instruction set and addressing possibilities not only to see which ones are available but also to understand why those choices were made and what the consequences of alternative choices would have been.

3.4. INSTRUCTION TYPES

Conventional machine level instructions can be approximately separated into two groups: general-purpose instructions and special-purpose instructions. The general-purpose instructions have wide application. For example, the ability to move data around in the machine is surely needed by almost every application. The special-purpose instructions have much narrower applications. For example, the Cyber 70 has an instruction that counts the number of 1's in an X register and leaves the result, an integer between 0 and 60, in another X register. Few applications can make effective use of this instruction. In the following sections we will discuss the major groups of general-purpose instructions.

Every instruction changes the state vector of the process that executed it. Even a NO OPERATION instruction changes the state vector because the program counter has a different value after it than before it. One can regard each instruction as a rule for transforming the old state vector into a new state vector. Some instructions change the value of one or more memory words or registers. Other instructions merely change

the program counter. In the following discussion of specific instructions keep in mind the fact that each instruction represents an algorithm for changing the old state vector into a new one.

3.4.1. Data Movement Instructions

The operation of copying data from one place to another is one of the most fundamental of all operations. By copying we mean the creating of a new object, with the identical bit pattern as the original. This use of the word "movement" is somewhat different from its normal usage in English. When we say that Marvin Mongoose has moved from New York to California, we do not mean that an identical copy of Mr. Mongoose was created in California and that the original is still in New York. When we say that the contents of memory location 2000 have been moved to some register, we almost always mean that an identical copy has been created there and that the original is still undisturbed in location 2000. Data movement instructions would better be called data duplication instructions, but the term data movement is already established.

Information can be stored in several places, differing in the way it is accessed. Three common places are in a particular memory word, in a register, or on the stack. The stack is kept either in special registers or in memory, but the way it is accessed is different from standard memory accesses. A memory access requires an address, whereas pushing an item onto the stack does explicitly address the stack. Data movement instructions require that both the source of the information (i.e., the original) and the destination (i.e., where the copy is to be placed) be specified either explicitly or implicitly.

Data movement instructions must somehow indicate the amount of data to be moved. Instructions exist to move quantities of data ranging from one bit to the entire memory. On fixed-word-length machines, the number of words to be moved is usually specified by the instruction—for example, separate instructions for move a word and move a half word. Variable-word-length machines often have instructions specifying only the source and destination addresses but not the amount. The move continues until an end of data field mark is found in the data itself.

The 370 has over 20 distinct instructions for moving data, the exact number depending on whether such instructions as load positive (LPR) are counted or not. Instructions are provided for moving data in units of 1, 2, 3, 4, and 8 bytes, as well as variable-length data of any length.

These instructions fall into four general groups: loading registers from memory, storing registers into memory, register-to-register transfer, and memory-to-memory moves. In addition, separate instructions are provided for transfers involving general registers, 32-bit floating-point numbers, and 64-bit floating-point numbers. Also, certain instructions not only move data but set the condition code (a two-bit register that the program can test), depending on whether the data is negative, positive, or zero. Such a large number of distinct instructions leads to a somewhat cumbersome instruction set.

It is often helpful, when trying to understand why certain decisions were made, to examine the alternative choices. An alternative design (not chosen by IBM) would have been to have a single, general-purpose move instruction. Such an instruction would have to specify the source of the data (general register, floating-point register, memory, or immediate operand), the destination of the data (general register, floating-point register, or memory), the number of bytes to be moved, and a bit telling whether the condition code was to be set or left unchanged.

An instruction of this kind is called **orthogonal** because each parameter is independent of the others. In addition to being simpler to use, such an instruction would be more general. For example, by specifying a move from memory to floating-point register 0 and a length of 32 bytes, all four floating-point registers could be loaded in one instruction. A single general-purpose move instruction, although easier to use, would have had the disadvantage of increasing the average instruction length, thereby using more memory and slowing down the machine. Machine designers are constantly forced to make compromises, trying carefully to balance all the relevant factors.

The Cyber 70 has fewer data types and, consequently, fewer instructions for moving them than the 370. There is one instruction to copy a 60-bit X register into another 60-bit X register, plus a group of instructions for moving 18-bit numbers from any register (A, B, or X) to any other register. The latter instructions are all special cases of instructions setting a register to the sum of a register and an 18-bit immediate constant in which the constant is zero.

All nine combinations of source (A, B, or X) and destination (A, B, or X) are provided. In addition, the instructions setting an A, B, or X register to the sum of an A register and a B register or the sum of two B registers reduce to data movement instructions when one of the registers is B0, which is always zero. No special instructions for loading registers from memory or storing registers into memory are provided because setting A1 through A5 automatically loads an X register from memory, and setting A6 or A7 automatically stores X6 or X7 into memory.

The PDP-11 has six data movement instructions. MOV and MOVB are orthogonal instructions that can move words and bytes respectively from general registers, memory, stacks, or immediate data to any of the above. The opcodes for MOV and MOVB differ only in one bit, bit 15, which can be regarded as the "length bit," specifying whether one or two bytes are to be moved. This orthogonal design was not carried over into the floating-point instructions (for lack of encoding room in the address specification field), and four separate instructions are provided for loading and storing 32- and 64-bit floating-point numbers.

3.4.2. Dyadic Operations

Dyadic operations are those that combine two operands to produce a result.

Just about all level 2 machines have an instruction to perform addition on integers. Except for very inexpensive minicomputers, subtraction, multiplication, and division of integers are standard as well. It is presumably unnecessary to explain why computers are equipped with arithmetic instructions.

Another group of dyadic operations includes the boolean instructions. Although there are 16 boolean functions of two variables, few, if any, level 2 machines have instructions for all 16. There is, for example, little need for the function that produces TRUE, independent of the arguments; if the bit pattern for TRUE is needed somewhere in the machine, it can simply be moved there rather than computed. Similar arguments suggest that an instruction to implement $f(P, Q) = P$ would be rather useless.

Three instructions present in many machines are AND, INCLUSIVE OR, and EXCLUSIVE OR. On fixed-word-length machines, AND computes the bit-by-bit boolean AND of two one-word arguments, the result of which is also a word. Sometimes there exist instructions for half words and/or double words as well. Similar remarks apply to the other operations.

A very important use of AND is for extracting bits from words. Consider, for example, a 32-bit word-length machine in which four 8-bit characters are stored per word. Suppose that it is necessary to separate the second character from the other three in order to print it; that is, it is necessary to create a word which contains that character in the rightmost 8 bits, referred to as **right justified** with zeros in the leftmost 24 bits.

To extract the character, the word containing the character is ANDed with a constant, called a **mask.** The result of this operation is that the unwanted bits are all changed into zeros—that is, masked out—as shown below.

```
10110111  10111100  11011011  10001011  A
00000000  11111111  00000000  00000000  B (mask)
--------  --------  --------  --------
00000000  10111100  00000000  00000000  A AND B
```

The result would then be shifted 16 bits to the right to isolate the character at the right end of the word.

A very important use of INCLUSIVE OR is to pack bits into a word, packing being the inverse of extracting. In changing the rightmost 8 bits of a 32-bit word without disturbing the other 24 bits, first, the unwanted 8 bits are masked out and then the new character is ORed in, as shown below.

```
10110111  10111100  11011011  10001011  A
11111111  11111111  11111111  00000000  B (mask)
--------  --------  --------  --------
10110111  10111100  11011011  00000000  A AND B
00000000  00000000  00000000  01010111  C
--------  --------  --------  --------
10110111  10111100  11011011  01010111  (A AND B) OR C
```

The AND operation tends to remove 1's, since there are never more 1's in the result than in either of the operands. The INCLUSIVE OR operation tends to insert 1's, since there are always at least as many 1's in the result as in the operand with the most 1's. The EXCLUSIVE OR operation, on the other hand, is symmetric, tending, on the average, neither to insert nor remove 1's. This symmetry with respect to 1's and 0's is occasionally useful—for example, in generating "random numbers."

Early computers performed floating-point arithmetic by calling software pro-

cedures, but today many computers, especially those intended for scientific work, have floating-point instructions at level 2 for reasons of speed. Some machines provide several lengths of floating-point numbers, the shorter ones for speed and the longer ones for occasions when many digits of accuracy are needed. Floating-point numbers are discussed in Appendix B.

3.4.3. Monadic Operations

Monadic operations have one operand and produce one result. Since one address less has to be specified than in the case of dyadic operations, the instructions are sometimes shorter.

Instructions to shift or rotate the contents of a word or byte are quite useful and are often provided in several different variations. Shifts are operations in which the bits are moved to the left or right, with bits shifted off the end of the word being lost. Rotates are shifts in which bits pushed off one end reappear on the other end. The difference between a shift and a rotate is illustrated below.

```
00000000 00000000 00000000 01110011   A
00000000 00000000 00000000 00011100   A shifted right 2 bits
11000000 00000000 00000000 00011100   A rotated right 2 bits
```

Both left and right shifts and rotates are useful. If an N-bit word is left rotated K bits, the result is the same as if it had been right rotated N − K bits.

Right shifts are often performed with sign extension. This means that positions vacated on the left end of the word are filled up with the original sign bit, 0 or 1. It is as though the sign bit were dragged along to the right. Among other things, it means that a negative number will remain negative. This situation is illustrated below.

```
11111111 11111111 11111111 11110000   A
00111111 11111111 11111111 11111100   A shifted right 2 bits without sign extension
11111111 11111111 11111111 11111100   A shifted right 2 bits with sign extension
```

An important use of shifting is multiplication and division by powers of 2. If a positive integer is left shifted K bits, the result is the original number multiplied by 2^K, barring overflow. If a positive integer is right shifted K bits, the result is the original number divided by 2^K.

Shifting can be used to speed up certain arithmetic operations. Consider, for example, computing 18 × N for some positive integer N. 18 × N = 16 × N + 2 × N. 16 × N can be obtained by shifting a copy of N four bits to the left. 2 × N can be obtained by shifting N one bit to the right. The sum of these two numbers is 18 × N. The multiplication has been accomplished by a move, two shifts, and an addition, which is often faster than a multiplication.

Shifting negative numbers, even with sign extension, gives quite different results, however. Consider, for example, the 1's complement number, −1. Shifted one bit to the left, it yields −3. Another one-bit shift to the left yields −7:

```
11111111 11111111 11111111 11111110   −1 in 1's complement
11111111 11111111 11111111 11111100   −1 shifted left 1 bit gives −3
11111111 11111111 11111111 11111000   −1 shifted left 2 bits gives −7
```

Left shifting 1's complement negative numbers does not multiply by 2. Right shifting does simulate division correctly, however.

Now consider a 2's complement representation of -1. Right shifted six bits with sign extension, it yields -1, whereas the integral part of $-\frac{1}{64}$ should be 0:

```
11111111 11111111 11111111 11111111    --1 in 2's complement
11111111 11111111 11111111 11111111    --1 shifted right 6 bits gives -1
```

Left shifting does, however, simulate multiplication by 2.

Rotate operations are useful in conjunction with packing and unpacking bit sequences from words. If it is desired to test all the bits in a word, rotating the word one bit at a time either way successively puts each bit in the sign bit, where it can be easily tested, and also restores the word to its original value when all bits have been tested.

The 370 has eight instructions for shifting data in registers. Two lengths are provided—32 bits and 64 bits—the latter consisting of two consecutive registers, which are treated as a unit. Both left and right shifts are provided. There are two types of shifts, arithmetic and logical. In the arithmetic shifts, the sign bit is unchanged and sign extension takes place for right shifts. In the logical shifts, all 32 or 64 bits are shifted, the sign bit may be changed, and no sign extension takes place. Shifts up to 64 bits may be specified for all eight shifts. There are no rotate instructions.

The Cyber 70 has four shift and rotate instructions, each of which operates on a 60-bit word in an X register. Two of them have the shift count built into the instruction, and two get the shift count from a B register. The arithmetic shift instructions extend the sign, and the rotate (or logical shift as CDC calls them) instructions rotate all 60 bits, including the sign bit.

The PDP-11 has ten shift and rotate instructions, eight of which form a group of one-bit shift and rotate instructions. Separate instructions are provided for word and byte, left and right, and shift and rotate. The other two (which are absent on some of the smaller models) act upon 16-bit or 32-bit data and can perform shifts or rotates of more than one bit. Unlike the 370 and Cyber, the PDP-11 has a carry bit that participates in most shift and rotate instructions. The carry bit is conceptually to the left of sign bit. A right rotate of one bit will move the carry bit into the sign bit and load the carry bit from the rightmost bit of the word or byte being rotated. This step facilitates multiple-precision arithmetic.

Certain dyadic operations occur so frequently with particular operands that level 2 machines sometimes have monadic instructions to accomplish them quickly. Moving zero to a memory word or register is extremely common when initializing a calculation. Moving zero is, of course, a special case of the general move data instructions. For efficiency, a CLEAR operation with only one address, the location to be cleared (i.e., set to zero), is often provided.

Adding 1 to a register or memory word is also commonly used for counting. A monadic form of the add instruction is the INCREMENT operation, which adds 1. The NEGATE operation is another example. Negating X is really computing 0 — X, a dyadic subtraction, but again, for efficiency, a separate NEGATE instruction is sometimes provided.

The PDP-11 has instructions to add 1 to a word or byte, subtract 1 from a word or byte, negate a word or byte, clear a word or byte, and compare a word or byte to 0, all unnecessary in the sense that a dyadic instruction could be used, but provided for reasons of efficiency. The instructions needed to add one word in memory, on the IBM 370, Cyber 70, and PDP-11, are shown below.

IBM 370	L	1, COUNTER	32 bits	load counter into register 1
	A	1, ONE	32 bits	add 1 to register 1
	ST	1, COUNTER	32 bits	store result back in counter
			96 bits	
CYBER 70	SA1	COUNTER	30 bits	load counter into X1
	SX6	X1 + 1	30 bits	add 1 to X1 (18 bit addition)
	SA6	A1	15 bits	store result in counter
			75 bits	
PDP-11	INC	COUNTER	32 bits	add 1 to counter

3.4.4 Comparisons and Conditional Jumps

Nearly all programs need the ability to test their data and alter the flow of control, based on the results. A simple example is the square root function, SQRT(X). If X is negative, the procedure gives an error message; otherwise it computes the square root. A program SQRT(X) has to test X and then branch, depending on whether it is negative or not.

A common method for doing so is to provide conditional jump (often called conditional branch) instructions that test some condition and jump to a particular memory address if the condition is met. Sometimes a bit in the instruction can be set to 1 or 0, meaning that the jump is to occur if the condition is met or the condition is not met, respectively.

The most common condition to be tested is whether a particular bit in the machine is 0 or not. If an instruction tests the sign bit of a 2's complement number and branches to LABEL if it is 1, the statements beginning at LABEL will be executed if the number was negative, and the statements following the conditional jump will be executed if it was 0 or positive. The same test made on a 1's complement number will always jump to LABEL if the tested number is less than or equal to -1, and it will never jump to LABEL if the tested number is greater than or equal to 1. If the number is 0, the jump may happen or not, depending on whether it is $+0$ or -0. That is obviously rather unpleasant, because mathematically $+0 = -0$, and is, in fact, one of the strongest arguments against 1's complement arithmetic and in favor of 2's complement arithmetic.

Many machines have bits that are used to indicate specific conditions. For example, the PDP-11 has an overflow bit that is set to 1 whenever an arithmetic operation gives an incorrect result. By testing this bit one checks for overflow on the previous arithmetic operation, so that if an overflow occurred, a jump can be made to an error routine. Similarly, some processors have a carry bit that is set when a carry

occurs from the leftmost bit—for example, if two negative numbers are added. A carry from the leftmost bit is quite normal and should not be confused with an overflow. Testing of the carry bit is needed for multiple-precision arithmetic.

Some machines have an instruction to test the rightmost bit of a word. It allows the program to test if a (positive) number is odd or even in one instruction.

Testing for zero is very important for loops and many other purposes. If all the conditional jump instructions tested only one bit, in testing a particular word for 0, one would need a separate test for each bit in order to ensure that none was a 1. To avoid this situation, many machines have an instruction to test a word and jump if it is zero.

Comparing two words or characters to see if they are equal or, if not, which one is greater is also important, in sorting for example. In performing this test, three addresses are needed, two for the data items and one for the address to jump to if the condition is true. Computers whose instruction format allows three addresses per instruction have no trouble, but those that do not must do something to avoid this problem.

One common solution is to provide a compare instruction that performs a comparison and that sets one or more condition bits to record the result. A subsequent instruction can test the condition bits and jump if the two compared values were equal, or unequal, or if the first was greater, and so on. The IBM 370 and PDP-11 both have a compare instruction that sets the condition code bits, plus conditional jump instructions that test them.

Some subtle points are involved in comparing two numbers. First, in 1's complement machines $+0$ and -0 are different, for example, on a 32-bit machine.

```
00000000 00000000 00000000 00000000    +0 in 1's complement
11111111 11111111 11111111 11111111    -0 in 1's complement
```

A decision has to be made by the designers whether $+0$ and -0 are equal and, if not, which is larger. No matter which way it is decided, there are convincing arguments that the wrong decision was made. If the designers make $+0 = -0$, then the fact that a comparison yields equal does not mean that the bit patterns of the compared data items are the same. Consider a machine with a 32-bit word in which four 8-bit characters are stored. If a word containing four identical characters whose bit representation is 0 is compared to a word containing four identical characters whose bit representation is 255—that is, the first word contains 32 zeros and the second 32 ones (as shown above)—the result will be that they are equal, which is not desired, since the characters are different.

If, on the other hand, the hardware treats -0 and $+0$ as unequal, then the result of adding -1 to $+1$ and comparing the result to $+0$ may very well be that they are unequal, since -1 added to $+1$ may give -0 as the result. Needless to say, this situation is highly undesirable. Two's complement machines, such as the 370 and PDP-11, do not have problems with $+0$ and -0, but one's complement machines, such as the Cyber 70, do have these troubles.

Another subtle point relating to comparing numbers is deciding whether or not the numbers should be considered signed or not. There are two ways of arranging the eight three-bit binary numbers in order, from smallest to largest.

Unsigned	Signed	
000	100	(smallest)
001	101	
010	110	
011	111	
100	000	
101	001	
110	010	
111	011	(largest)

The column on the left shows the positive integers 0 to 7 in increasing order. The column on the right shows the 2's complement signed integers -4 to $+3$. The answer to the question "Is 011 greater than 100?" depends on whether or not the numbers are regarded as being signed.

Machines with fewer than three addresses per instruction sometimes handle conditional jumping with an instruction that skips the next sequential instruction if the condition is met. That instruction will frequently be a jump instruction. On some machines several instructions may be skipped, instead of just one, the number to be skipped being specified in the instruction itself. The PDP-11, for example, can skip up to 127 words forward or 128 words backward. Whether one considers this a glorified skip instruction or an emaciated conditional jump instruction is a matter of taste.

The unconditional jump is a special case of the conditional jump in which the condition is always met.

3.4.5. Procedure Call Instructions

A procedure is a group of instructions that performs some task and that can be invoked (called) from several places in the program. The term **subroutine** is often used instead of procedure. When the procedure has finished its task, it must return to the statement after the call. Therefore the return address must be transmitted to the procedure so that it knows where to return to.

There are three places the return address may be placed: memory, a register, or the stack. Far and away the worst solution is putting it in a single, fixed-memory location. In this scheme, if the procedure called another procedure, the second call would cause the return address from the first one to be lost.

A slight improvement is having the procedure call instruction store the return address in the first word of the procedure, with the first executable instruction being in the second word. The procedure can then return by jumping indirectly to the first word or, if the hardware puts the opcode for jump in the first word along with the return address, jumping directly to it. The Cyber 70 uses this method. The procedure may call other procedures, since each procedure has space for one return address. If

the procedure calls itself, this scheme fails, for the first return address will be destroyed by the second call. The ability for a procedure to call itself, known as **recursion,** is exceedingly important both for theorists and practical programmers. Furthermore, if procedure A calls procedure B, and procedure B calls procedure C, and procedure C calls procedure A, known as **daisy chain recursion** (a reference to a closed bracelet made out of daisies), this scheme also fails.

A bigger improvement is to have the procedure call instruction put the return address in a register, leaving the responsibility for storing it in a safe place to the procedure. If the procedure is recursive, it will have to put the return address in a different place each time it is called. The 370 uses this method.

The best thing for the procedure call instruction to do with the return address is to push it onto a stack. When the procedure has finished, it pops the return address off the stack and stuffs it into the program counter. If this form of procedure call is available, then recursion does not cause any special problems: the return address will automatically be saved in such a way as to avoid destroying previous return addresses. The PDP-11 uses this method.

3.4.6. Loop Control

The need to execute a group of instructions a fixed number of times occurs frequently, and thus some machines have instructions to facilitate doing this. All the schemes involve a counter that is increased (or decreased) by some constant once each time through the loop. The counter is also tested once each time through the loop. If a certain condition holds, the loop is terminated.

For example, one looping method involves initializing a counter to N, the number of times the loop is to be executed, before the loop is begun. The last instruction of the loop subtracts 1 from the counter and, if the result is greater than zero, jumps back to the first instruction of the loop. Otherwise the loop is finished, and it falls through, executing the next instruction. This form of looping is characterized as test-at-the-end type looping, and it is shown in Fig. 3-32(a).

Test-at-the-end looping has the property that the loop will always be executed at

```
      COUNTER = N;
LOOP: /* FIRST STATEMENT */
      /* SECOND STATEMENT */
          .
          .
          .
      /* LAST STATEMENT */
      COUNTER = COUNTER - 1;
      IF COUNTER >0 THEN GO TO LOOP;
EXIT: /* FIRST STATEMENT AFTER LOOP */
```

(a)

```
      COUNTER = N;
LOOP: IF COUNTER< = 0 THEN GO TO EXIT;
      /* FIRST STATEMENT */
      /* SECOND STATEMENT */
          .
          .
          .
      /* LAST STATEMENT */
      COUNTER = COUNTER - 1;
      GO TO LOOP;
EXIT: /* FIRST STATEMENT AFTER LOOP */
```

(b)

Fig. 3-32. (a) Test-at-the-end looping. (b) Test-at-the-beginning looping.

least once, even if N is less than or equal to 0. Consider as an example, a program that maintains personnel records for a company. At a certain point in the program, it is reading information from cards about a particular employee. It reads in N, the number of children the employee has, and executes a loop N times, once per child, reading the child's name, sex, and birthday, so that the company can send him or her a birthday present, one of the company's fringe benefits. If the employee does not have any children, N will be 0, but the loop will still be executed once, sending presents and giving erroneous results.

Figure 3-32(b) shows another way of performing the test that works properly even for N less than or equal to 0. Notice that the testing is different in the two cases, so that if a single level 2 instruction does both the decrement and the test, the designers are forced to choose one method or the other. The PDP-11 has an instruction that subtracts 1 from a counter and branches if the result is nonzero. The 370 has a more general loop instruction that can add an arbitrary positive or negative constant instead of only 1. The Cyber 70 has no loop instruction.

3.4.7. Input/Output

No other group of instructions exhibits as much variety from machine to machine as the i/o instructions. A basic distinction can be made between level 2 machines that have explicit i/o instructions and those that do not.

The simplest possible i/o method is the one used in some minicomputers. These minicomputers have a single i/o instruction that selects one of the i/o devices. A single character is transferred between a fixed register in the processor and the selected i/o device. The processor must execute an explicit instruction for each and every character read or written.

The next step upward is a computer in which the CPU specifies the device, the memory address where the data is or is to be put, and the number of words or characters to be transmitted. Such a computer allows one instruction per block of data. The PDP-11 disk transfers are of this form.

The 370 is an example of a computer with a simple i/o processor, which IBM calls a channel. The IBM channel is a special-purpose computer with a limited instruction set consisting of read, write, read backward, control (e.g., rewind a tape, skip a line on the printer) sense (provides status information), and jump. The CPU performs i/o by first creating a channel program and storing it in main memory, assuming the channel program was not already there. Then it puts the address of the channel program in the word at main memory location 72. Finally, the CPU executes a START I/O instruction whose address field specifies the channel and i/o device to be started. The channel then fetches the address of its program from memory location 72, puts that address in its program counter, and begins executing its channel program. The various memory words involved are illustrated in Fig. 3-33.

A channel program consists of one or more 64-bit instructions for the channel. Each instruction contains an 8-bit opcode telling which operation is to be performed

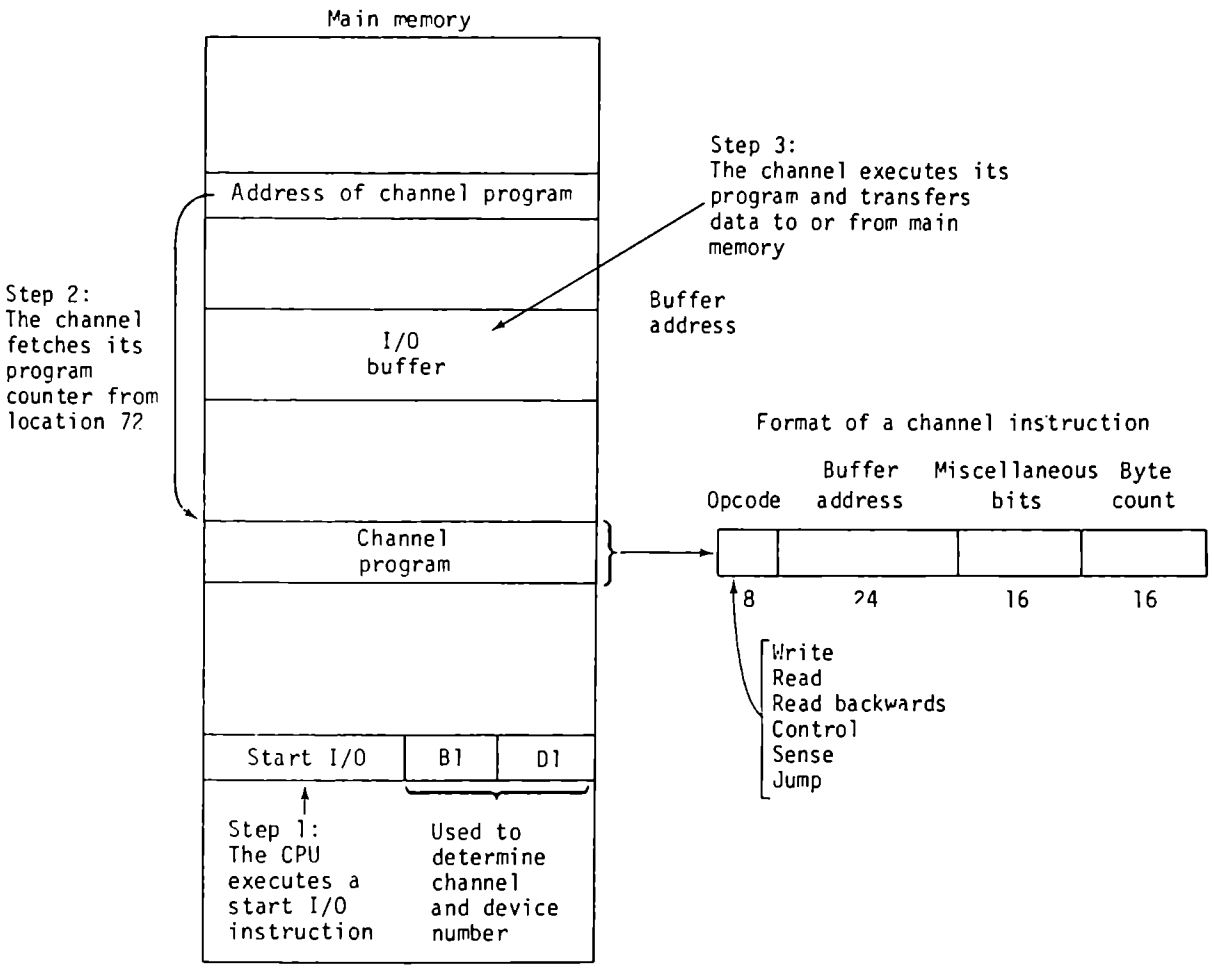


Fig. 3-33. The steps involved in performing i/o at the conventional machine level on the IBM 370.

(e.g., read, write, jump), a 24-bit main memory address, telling where the data is to be read from or written into, and a count, telling how many bytes are to be transferred. In addition, some miscellaneous bits specify such items as no data transmission (good for skipping a record on a tape) and "stop the channel after this instruction is complete." Once the START I/O instruction has been executed, the CPU is free to continue computing in parallel with the channel.

In addition to the START I/O instruction, the 370 CPU has a few other i/o instructions. The HALT I/O instruction forcibly stops all activity in the selected channel. The TEST I/O and TEST CHANNEL instructions are used to determine the current status of i/o activity.

The group of level 2 machines that has no explicit i/o instructions can be sub-

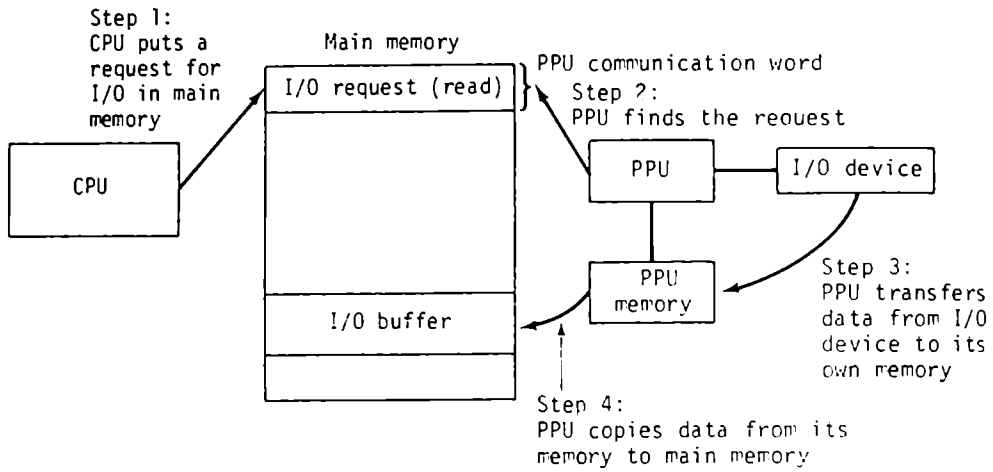


Fig. 3-34. The steps involved in performing i/o on the Cyber 70.

divided into a group that really cannot perform any i/o at all, a sort of blind deafmute computer, and a group that can perform i/o without the use of explicit instructions.

The Cyber 70 conventional machine level is an example of a computer without any i/o capability at all. To communicate with the outside world, the CPU must rely on help from its i/o processors, the PPUs. When the CPU wants some i/o performed, it leaves a message in main memory. Eventually one of the PPUs finds the message and carries out the required i/o operation (Fig. 3-34).

The PDP-11 level 2 machine can perform i/o without the use of explicit i/o instructions. Each i/o device has one or more status registers that control its behavior, plus one buffer register through which all information passes on its way into or out of the device. Every status and buffer register for each device has a unique memory address that the CPU can directly address. Since the device registers are addressable as ordinary memory locations, all the usual data movement and testing instructions can operate on them and no special i/o instructions are needed.

As an example of the operation of the PDP-11 i/o system, consider the paper tape reader. Associated with the paper tape reader are two registers: the paper-tape-reader status register, at address 777550 (octal), and the paper-tape-reader buffer register, at address 777552 (octal). These registers are shown in Fig. 3-35(a). Five of the bits in the status register are used; the other 11 are ignored. When bit 0 is set to a 1, by the MOV, INC, or ADD instruction (among others), the reader begins reading a character. When read, the character is placed in the low-order byte of the buffer register. Because the buffer register has a memory address, the character can be fetched by using a MOVE BYTE instruction, thus eliminating the need for a special i/o instruction.

More complicated i/o devices require more status registers. The RK05/RK11 disk, for example, uses a total of six registers containing 43 distinct fields of information, as shown in Fig. 3-35(b). Each register has a unique memory address by which the program can read it or store into it.

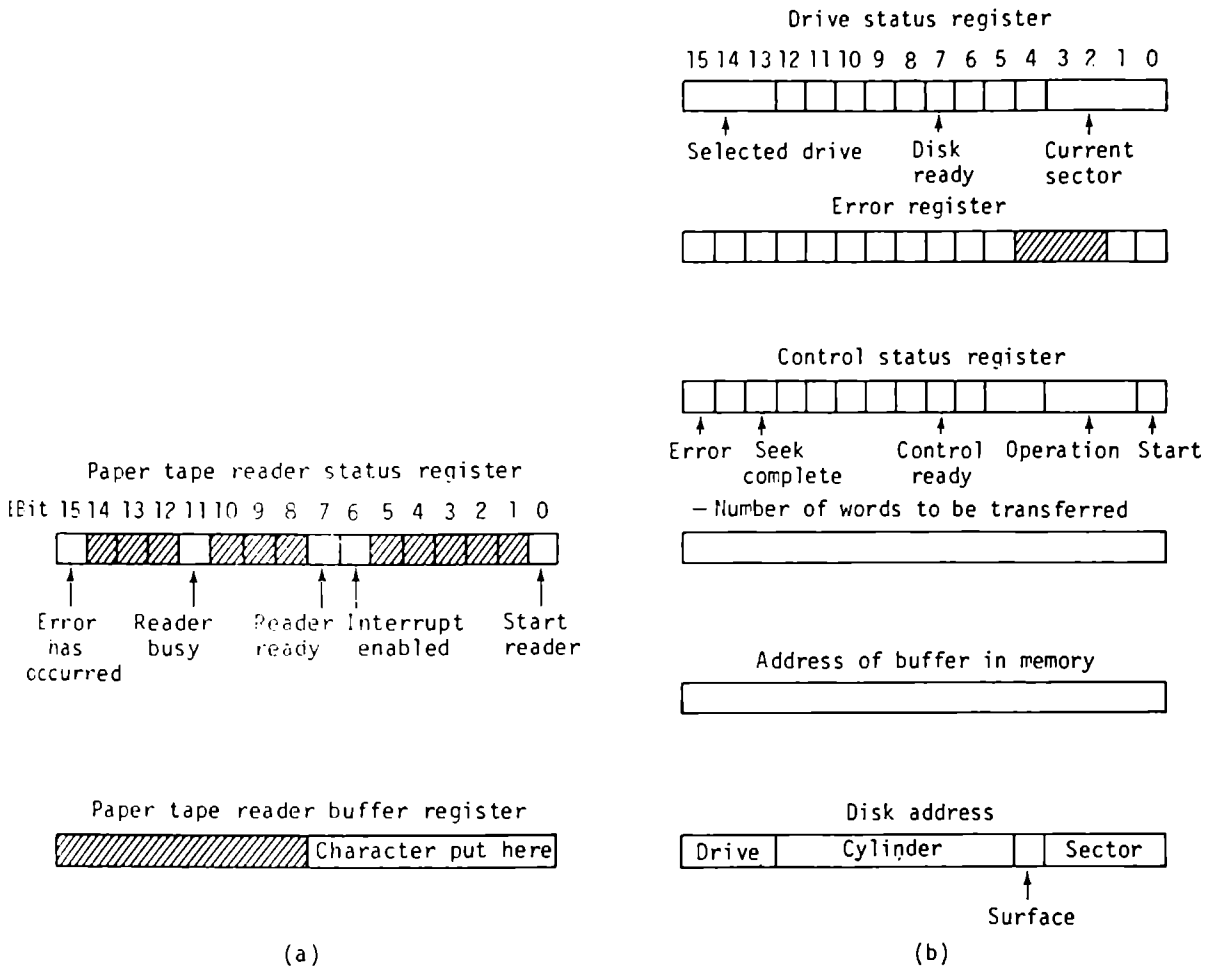


Fig. 3-35. PDP-11 conventional machine level programs can directly reference the i/o status and buffer registers the same way they can reference memory. (a) The paper tape reader registers. (b) The registers used by the RK11 disk controller. Most of the 43 fields relate to error conditions or the status of the hardware.

3.5. REPRESENTATION OF DATA

Almost all the instructions discussed earlier perform some operation on the data stored in the computer's memory and thereby change the state of the computation. An operation that did not change the process' state would not be a very interesting one. We will now discuss the various kinds of primitive data items which level 2 machines can manipulate, the ways that more complex data is built up from simpler data, and the ways in which data is represented within the memory.

The representation of data is highly machine dependent. In particular, there is a great difference between machines with a fixed word length and machines with no

words at all. Since the vast majority of present-day machines have fixed word lengths, the discussion will focus on them. It should be noted that the main reason fixed-word-length machines are so popular is that transfers and operations can be performed in parallel, thus greatly speeding up the operation of the machine. The IBM 370, Cyber 70, and PDP-11 all have fixed word lengths. On these machines, when a word is moved from one location to another, all the bits are moved in parallel. This situation is in contrast to a serial machine, in which bits or characters are moved one at a time from left to right or right to left.

3.5.1. Integers

Integers are the objects whose representation is the most standardized. An integer is usually one machine word, using 1's complement or 2's complement notation for negative numbers. Double-precision integers can be stored in two consecutive machine words. Integers are available as a data type in almost all programming languages and are widely used in algorithms of all sorts. Integers are used for counting (e.g., number of pelicans left), as array subscripts (e.g., the twelfth president taken from a list of the presidents), in number theory (finding prime numbers), and many other places.

3.5.2. Floating-Point Numbers

Floating-point numbers have two important advantages over integers. First, their range is much larger than that of integers. For the IBM 370, Cyber 70, and PDP-11, the approximate value of the largest real number is 10^{76}, 10^{322}, and 10^{38}, respectively. This property is useful in problems dealing with numbers that span a wide range of magnitudes. In quantum physics, for example, numbers as small as 9×10^{-28} (mass of the electron in grams) and as large as 3×10^{10} (speed of light in cm/sec) often occur together.

Second, floating-point numbers are useful for representing numbers with fractional parts. Despite the fact that the Indiana State Legislature once decreed π to be exactly 3, for most calculations 3.1416 is a better approximation.

3.5.3. Booleans

Booleans (or one-bit strings) are variables that can take on one of two values, TRUE or FALSE. One bit is needed to represent a boolean. One possible method is to store TRUE as 1 and FALSE as 0. On a fixed-word-length machine, a whole word is wasted for one bit. But this is not the worst of it. Representations for data should be chosen to simplify the basic operations to be performed on the data. For booleans, the basic operations are testing to see if a boolean is TRUE or FALSE, complementing—that is, boolean NOT—and the dyadic operations AND, OR, and EXCLUSIVE OR. Unless the machine has an instruction to test the rightmost bit of a word, testing becomes difficult, which is a severe drawback of this representation. Using the sign

bit instead of the low-order bit would be better, for most machines can test the sign of a word easily.

Another possibility would be to use a word of 0's for FALSE and a word of 1's for TRUE. In 1's complement this would be FALSE = −0, and TRUE = +0; and in 2's complement this would be FALSE = −1 and TRUE = 0. If the machine has an instruction to complement a word bit by bit (i.e., 1's complement), which will certainly be present on 1's complement machines and probably on 2's complement machines as well (the Cyber 70 and PDP-11 have one, but the 370 does not), boolean NOT becomes easy. In the method of using 0 and 1, the bit-by-bit complement instruction operating on 0 does not produce 1, but rather an entire word full of 1's, so that method is not closed with respect to the word-complementing instruction.

Yet another possibility is to use 0 for FALSE and everything else for TRUE. This method has the advantage of making tests for TRUE and FALSE easy, but the boolean operators more complicated.

3.5.4. Characters

If the number of bits per memory cell is (nearly) the same as the character size, it makes sense to store characters one per cell and there are few problems. For example, on the IBM 370 and PDP-11, characters are usually stored one per byte. If, however, the word size is several times larger than the character size, as is the case on the Cyber 70 with a 60-bit word and a 6-bit character, then one must decide if characters are to be left justified or right justified. On the Cyber 70, one can either put the character in bits 0 to 5—that is, right justified with 54 unused bit positions to the left of the character—or in bits 54 to 59, with the unused bits to the right of the character. Furthermore, one must decide what to do with the unused bits. One possibility is to fill them all with zeros.

3.5.5. Strings

Strings of characters are commonly used to store text read in or text to be printed out later. There are several methods for storing the characters internally, with the choice of method depending on what operations are to be performed most often on the strings. The simplest method, and the most frugal in terms of space, is packing as many characters per word as will fit, with the words stored consecutively. Figure 3-36 shows this for the string

```
IF X>0 THEN CALL OK;
```

for a machine with four characters per word. Sometimes the representation of Fig. 3-36(a) is more convenient, and sometimes that of Fig. 3-36(b) is more convenient.

This representation, although very compact, does have a disadvantage: namely, if it is necessary to insert or delete characters from the middle of the string, one must alter every word following the change. For example, users of time-sharing systems keep their programs in the computer. There is a text-editing program that allows them

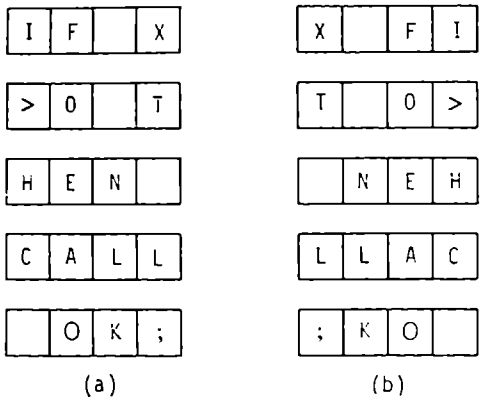


Fig. 3-36. A character string stored four characters per word. (a) Stored left to right. (b) Stored right to left.

to make changes to the source text. If a user wants to add a new statement at the beginning of his source program and it is stored as a packed string as in Fig. 3-36, the text editor has to move the whole program to make room for it. This is a time-consuming operation.

Another method for storing strings is one character per word, with part of each word containing a pointer giving the address of the next character. The words need not be in order. Figure 3-37(a) illustrates this situation, with the character left justified and the pointer right justified. Such a representation is one kind of a **linked list.** Word 1000 contains the first letter, I, and a pointer to word 1001, which contains the second letter and a pointer to the word containing the third letter, and so on. The end of the string must be marked somehow—for example, by having 0 as a pointer.

In this form, inserting text in the middle is quite easy. Figure 3-37(b) shows the string changed to

```
IF SIZE>0 THEN CALL OK;
```

new machine words have been added at the end, containing the letters I, Z, and E. To insert them into the list, the pointer in word 1003 had to be changed, but nothing had to be moved. Storing strings as linked lists wastes a lot of space compared with packing the characters four to a word, but it saves the time needed to move a lot of text around.

A variation on the latter method is storing a string as a list of blocks, each of which contains several machine words with four characters per word and one word as a pointer to the next block. Adding new text in the middle requires changing one block, not the whole string. For example, source text from cards might be stored in blocks of 21 words, 20 words of four characters preceded by a word pointing to the next block. Adding a new source statement would require changing only one pointer; and even modifying an existing statement would require, at most, moving 20 words, no matter how long the string (i.e., the source program) was. Yet the storage utilization is only 5 percent worse than the first method, since 1 word in 21 does not contain data.

Still another variation is to use variable-sized blocks, with the block length stored in the pointer word. Trailing blanks on each card could be ignored, thus saving much

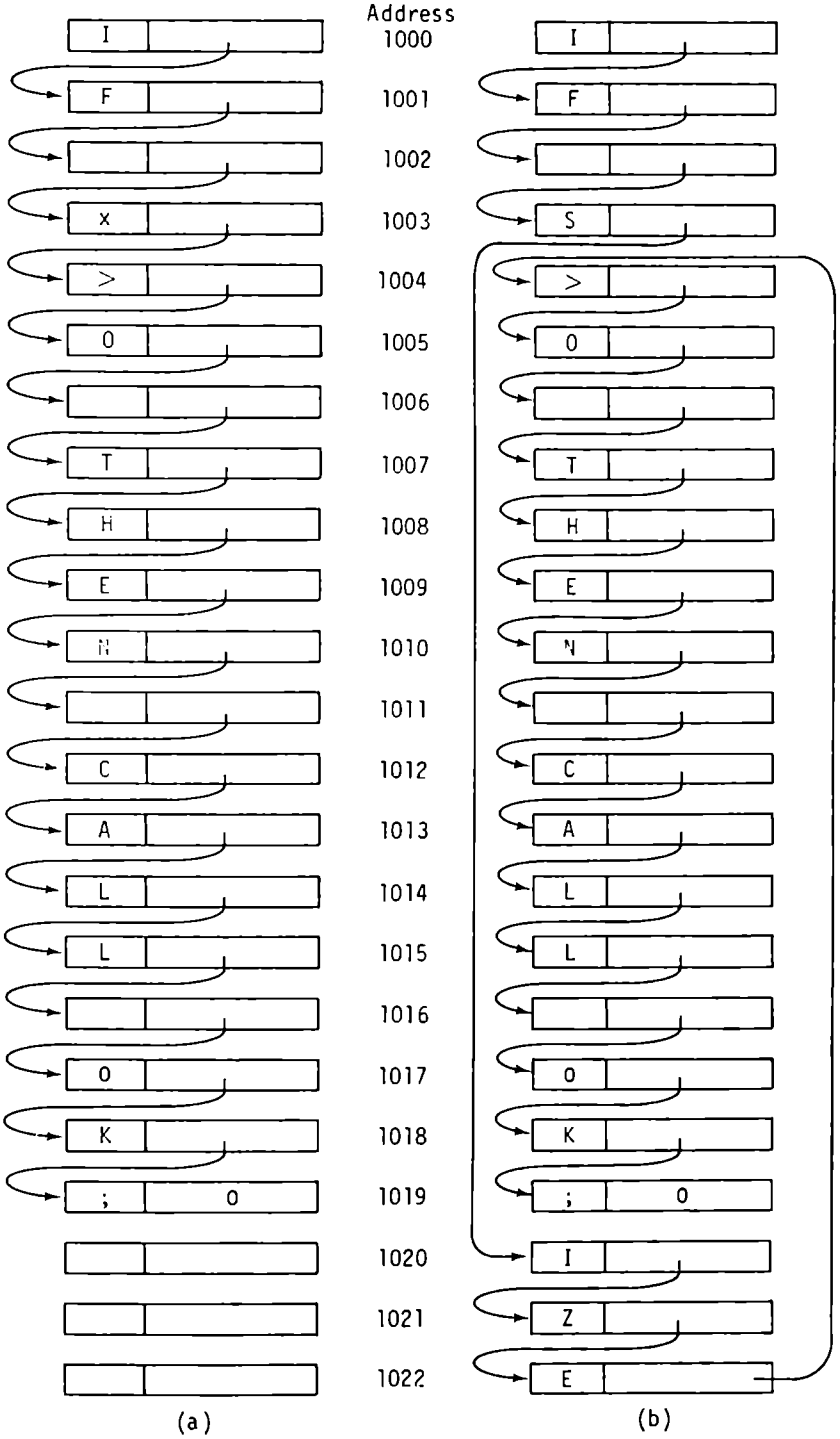


Fig. 3-37. Storing a string as a linked list. (a) Original string. (b) After changing X to SIZE.

space, for statements in a program rarely fill up even half of a card. Of course, with ordinary English text, which used all 80 columns, variable-sized blocks might not be worth the effort required to manipulate them.

3.5.6. Arrays

Vectors, matrices, and three-dimensional arrays have many applications and are widely used in computer science. The elements of an array may be integers, floating-point numbers, booleans, characters, or more complex objects built up from these types. This section will discuss representation of arrays in terms of integers, but the methods given are equally useful for the other data types as well.

Dope Vectors

Consider a three-dimensional array A as an example. The first subscript, i, must be in the range $l_1 \leq i \leq u_1$. The second subscript, j, must be in the range $l_2 \leq j \leq u_2$, and the third subscript, k, must be in the range $l_3 \leq k \leq u_3$. If each element is an integer occupying one word, the number of words required to store the array is

$$(u_3 - l_3 + 1) \times (u_2 - l_2 + 1) \times (u_1 - l_1 + 1).$$

One storage method keeps $A(i, j, k)$ at

$$a + (k - l_3) \times (u_2 - l_2 + 1) \times (u_1 - l_1 + 1) + (j - l_2) \times (u_1 - l_1 + 1) + (i - l_1)$$

where a is the address of A. Note that $A(i, j, k)$ and $A(i + 1, j, k)$ are stored in consecutive words. Some algebra reveals that $A(i, j, k)$ is stored at

$$d_0 + i \times d_1 + j \times d_2 + k \times d_3$$

where

$$d_0 = a - l_3 \times (u_2 - l_2 + 1) \times (u_1 - l_1 + 1) - l_2 \times (u_1 - l_1 + 1) - l_1$$
$$d_1 = 1$$
$$d_2 = u_1 - l_1 + 1$$
$$d_3 = (u_2 - l_2 + 1) \times (u_1 - l_1 + 1)$$

The constants d_i depend only on the upper and lower limits and therefore can be computed once and for all as soon as the limits are known. They are sometimes called **strides**, because they control the interval between elements, one of whose subscripts differs by 1. Since checking to ensure that subscripts are within range is highly desirable in order to catch errors, it is necessary to store the limits and the strides for use during execution of the program, using the arrays.

Figure 3-38 shows a little table that is stored in the memory and used whenever the address of $A(i, j, k)$ must be calculated. Such a little table is called a **dope vector**, because it gives you all the "dope" about the array.

In order to perform the subscripting, six comparisons must be performed—namely, i and l_1, i and u_1, j and l_2, j and u_2, k and l_3, and k and u_3—to see if all subscripts are within the allowed range. One can easily imagine a machine on which all six comparisons could be done simultaneously along with the three required multipli-

Offset (d_0)		
Upper bound for i (u_1)	Lower bound for i (l_1)	Multiplier for i (d_1)
Upper bound for j (u_2)	Lower bound for j (l_2)	Multiplier for j (d_2)
Upper bound for k (u_3)	Lower bound for k (l_3)	Multiplier for k (d_3)

Fig. 3-38. Dope vector for locating $A(i,j,k)$.

cations—$i \times d_1$, $j \times d_2$, and $k \times d_3$. Assuming that a multiplication takes longer than a comparison, the total time for the three multiplications and six comparisons would be one multiplication time. This would be followed by adding d_0 to $i \times d$ and adding $j \times d_2$ to $k \quad d_3$ in parallel. This would be followed by adding those two sums together.

The complete subscripting, including bounds checking for a three-dimensional array, would be one multiplication time plus two addition times. Unfortunately, few computers have the ability to perform the required operations in parallel, even though the cost of the integrated-circuit chip needed is negligible compared to the total cost of the machine. As a result, subscripting three-dimensional arrays takes 10 times as long as it should.

Marginal Indexing

We will now examine a totally different way of storing arrays. It is easiest to illustrate this method by an example. Consider a three-dimensional array, A, declared by

```
DECLARE A(1:2,1:3,1:4) FIXED BINARY
```

The left side of Fig. 3-39 shows a table with two pointers, corresponding to the two values i may take on: 1 and 2. Each of these pointers points to a three-word table, corresponding to the three values j may take on. These, in turn, point to the data.

In order to find $A(i, j, k)$, i is used as an index into the leftmost table to get a pointer to a second table. Then j is used as an index into the second table to get a pointer to a third table. Each block in the third table contains the data for $A(i, j, 1)$, $A(i, j, 2)$, $A(i, j, 3)$, and $A(i, j, 4)$. By using k as an index, the proper element can be found.

Marginal indexing is obviously wasteful of space compared to the dope vector approach, since the first two tables for i and j must be stored. It does have the virtue, however, that no multiplications are needed; and on a level 2 machine with no integer

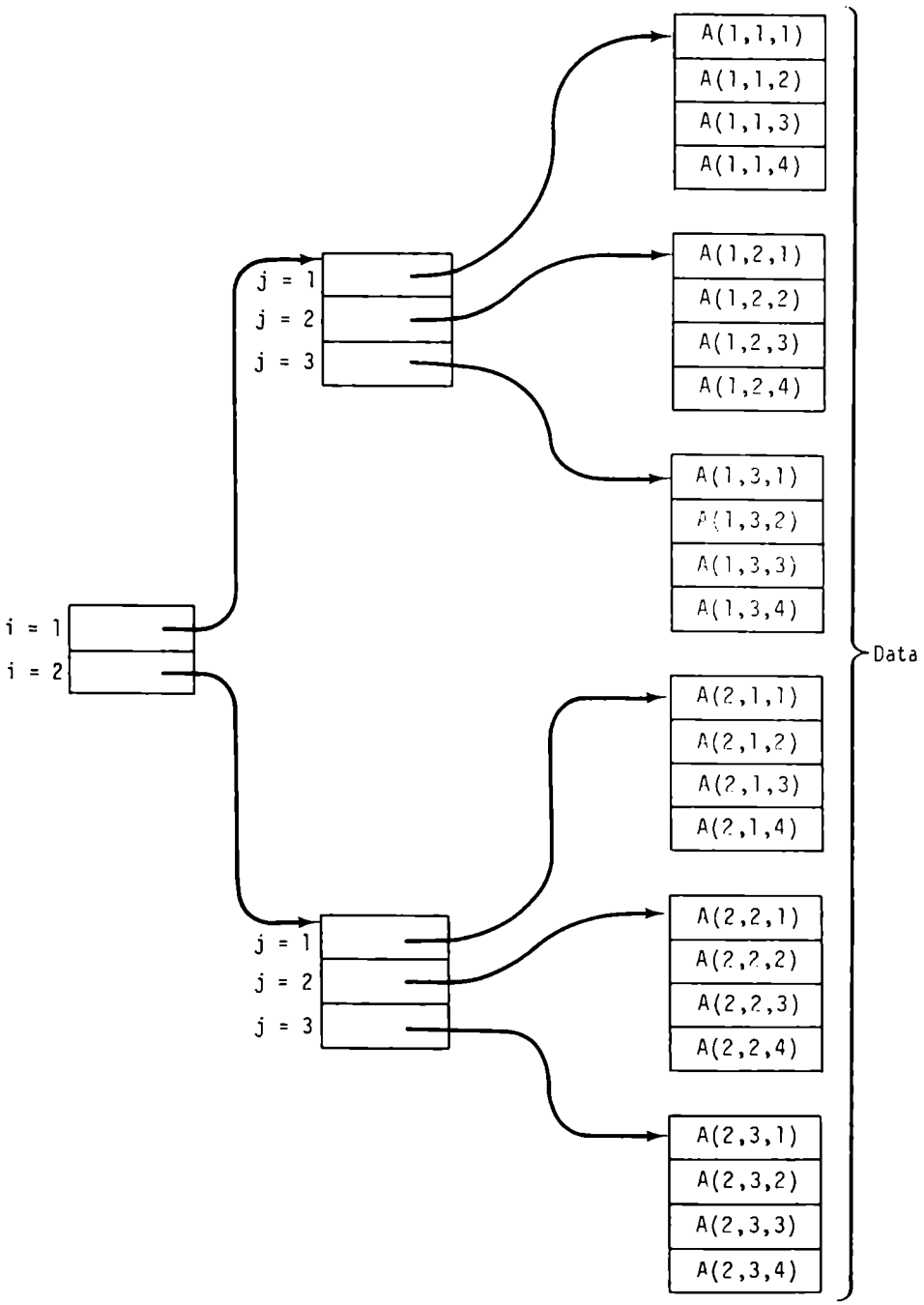


Fig. 3-39. Subscripting using marginal indexing.

multiplication instruction, or a very slow one, this method may be faster than using the dope vector approach.

Marginal indexing is also useful for working with large, sparse arrays. A sparse array is one in which most elements are 0. By not creating data blocks containing only zeros, considerable memory can be saved.

3.6. FLOW OF CONTROL

Flow of control refers to the sequence in which instructions are executed. In general, successively executed instructions are fetched from consecutive memory locations. Procedure calls cause the flow of control to be altered, stopping the procedure currently executing and starting the called procedure. Coroutines are related to procedures and cause similar alterations in the flow of control. Traps and interrupts also cause the flow of control to be altered when special conditions occur. All these topics will be discussed in the following sections.

3.6.1. Sequential Flow of Control and Jumps

Most instructions do not alter the flow of control. After an instruction is executed, the one following it in memory is fetched and executed. After each instruction, the program counter is increased by the number of memory locations in that instruction. If observed over an interval of time that is long compared to the average instruction time, the program counter is approximately a linear function of time, increasing by the average instruction length per average instruction time. Stated another way, the dynamic execution sequence, the order in which the processor actually executes the instructions, is the same as the order in which they appear on the program listing.

If a program contains jumps (branches), this simple relation between the order in which instructions are written and the order in which they are executed is no longer true. When jumps are present, the program counter is no longer a monotonically increasing function of time, as shown in Fig. 3-40(b). As a result, it becomes difficult to visualize the instruction execution sequence from the program listing. When programmers have trouble keeping track of the sequence in which the processor will execute the instructions, they are prone to make errors. This observation led Dijkstra (1968a) to write a then controversial letter entitled "GOTO Statement Considered Harmful," in which avoiding GOTO statements is suggested. Since that time, programming languages without GOTOs have become popular. At level 2, GOTOs are usually called jumps or branches.

Jumps are frequently difficult to avoid when programming in a language lacking structuring statements, such as IF . . . THEN . . . ELSE and DO WHILE. Relying on jumps as the primary method of controlling the flow of execution makes it difficult, if not impossible, to write error-free, well-structured programs. For this and other reasons, programming at the conventional machine level is becoming obsolete, although operating systems and other programs whose function is to support higher levels are still written for this level (or compiled to this level), of course.

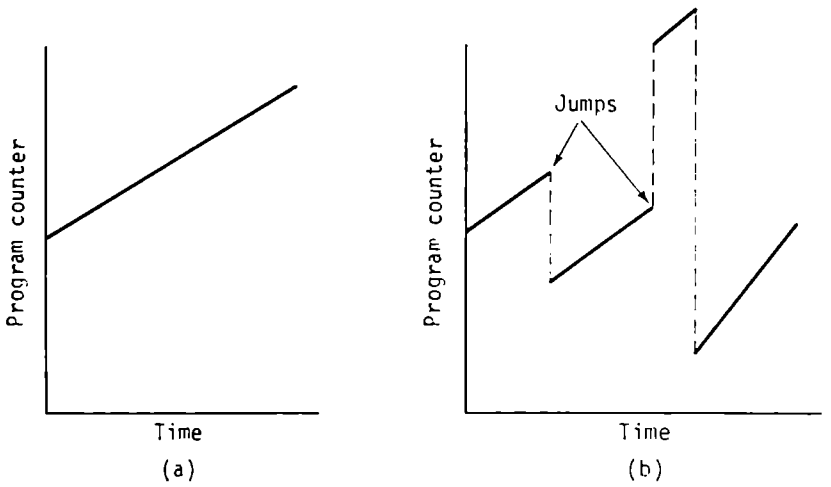


Fig. 3-40. Program counter as a function of time (smoothed). (a) Without jumps. (b) With jumps.

3.6.2. Procedures

The most important technique for structuring programs is the procedure. From one point of view, a procedure call alters the flow of control just as a jump does, but unlike the jump, when finished performing its task, it returns control to the statement or instruction following the call.

However, from another point of view, a procedure body can be regarded as defining a new instruction on a higher level. From this standpoint, a procedure call can be thought of as a single instruction even though the procedure may be quite complicated. Similarly, a person programming at level 2 can certainly regard the multiplication instruction as a single instruction even though he is well aware that it is carried out by an interpreter running at level 1 as a large number of successive steps.

By writing a collection of procedures, a programmer can define a new level with a new, larger, and more convenient instruction set. Programs for this new level consist of sequences of instructions, some of which are procedure calls and same of which are the original level 2 instructions. Associated with the execution of a program at this new level is a "virtual program counter," which points to the current instruction (counting a procedure execution as a single instruction) and increases monotonically in time. The direct correspondence between the execution sequence and the listing sequence makes it easy to understand what the program does.

In Section 3.4.5 we mentioned recursive procedures—that is, procedures that call themselves, possibly via a daisy chain. Now we will give an example of one. The "Towers of Hanoi" is an ancient problem that has a simple solution involving recursion. There are three pegs, the first of which passes through the center of a series of N concentric disks, each of which is smaller in diameter than the disk directly below it. The second and third pegs are initially empty. The object is to transfer all the

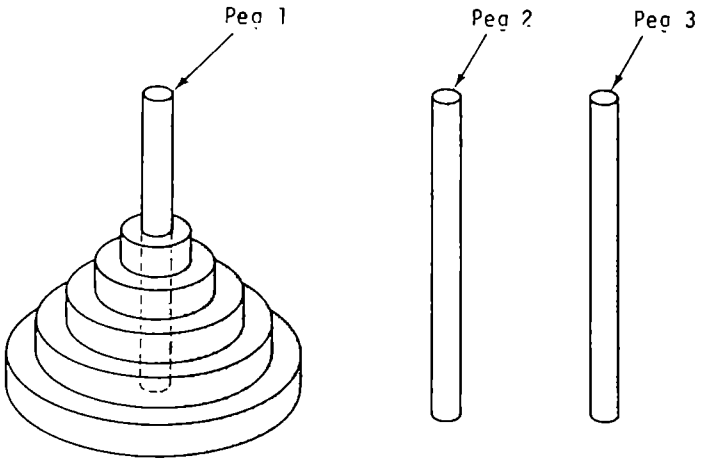


Fig. 3-41. Initial configurations for the Towers of Hanoi problem for five disks.

disks to peg 3, one disk at a time, but at no time may a larger disk rest on a smaller one. Figure 3-41 shows the initial configuration for N = 5 disks.

The solution of moving N disks from peg 1 to peg 3 consists first of moving N − 1 disks from peg 1 to peg 2, then moving one disk from peg 1 to peg 3, and then moving N − 1 disks from peg 2 to peg 3 (see Fig. 3-42). To solve the problem, we need a procedure to move N disks from peg I to peg J. Whenever this procedure is called, by

```
CALL TOWERS (N,I,J)
```

the solution is printed out. The procedure first tests to see if N = 1. If so, the solution is trivial; just move it from I to J. If N ≠ 1, the solution consists of three parts as discussed above, each being a recursive procedure call. The complete solution is shown in Fig. 3-43. The call

```
CALL TOWERS (3,1,3)
```

to solve the problem of Fig. 3-42, will generate three more calls, since N is not 1, namely,

```
CALL TOWERS (2,1,2)
CALL TOWERS (1,1,3)
CALL TOWERS (2,2,3)
```

The first and third will, in turn, also generate three more calls, making seven generated calls in all.

In order to have recursive procedures at level 2, we must use a modified form of a stack to store the parameters. Each time a procedure is called, a block of memory called a **stack frame** is reserved on the stack for the return address, parameters, and local variables, if any. The frame most recently created is the current frame. On some

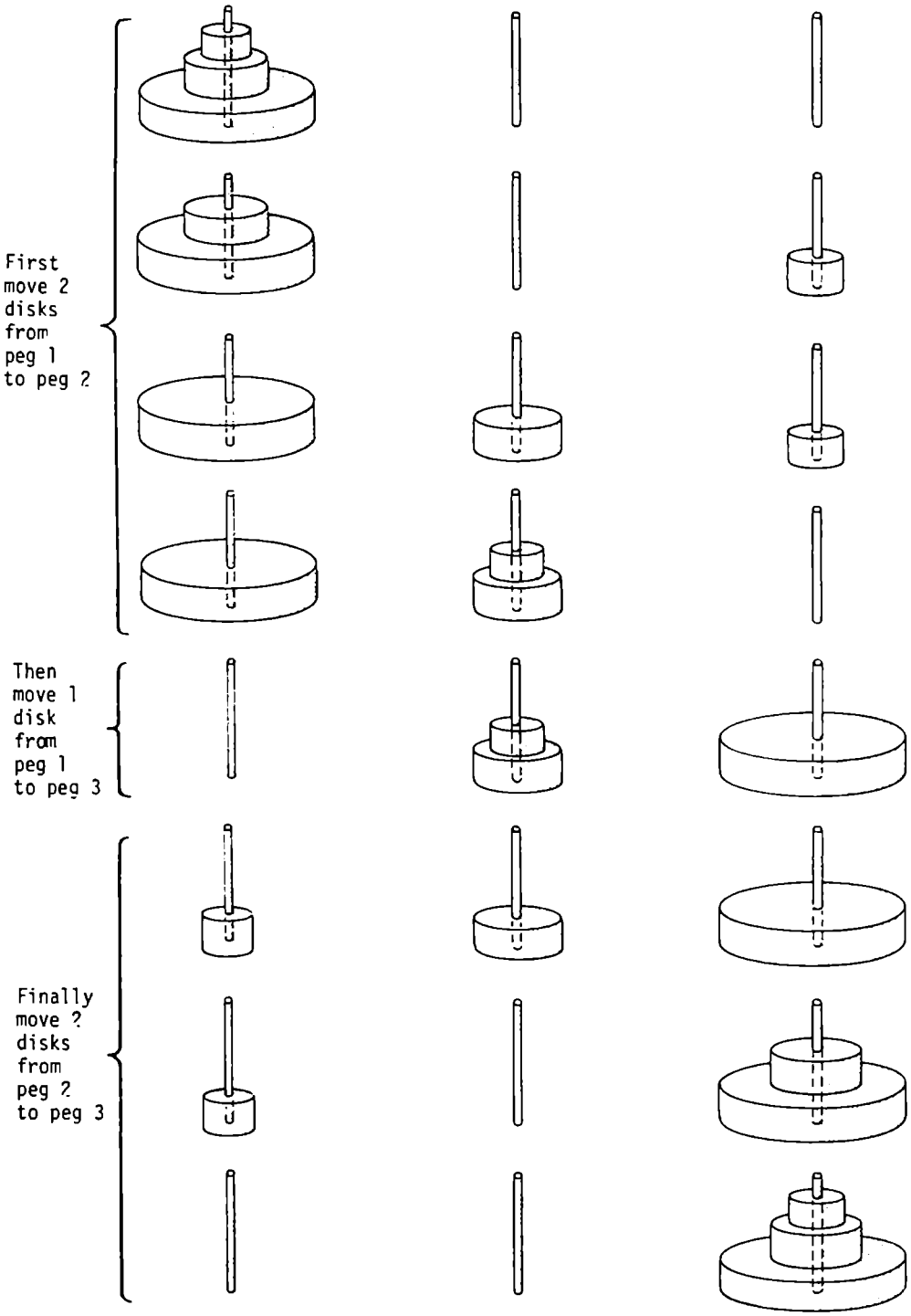


Fig. 3-42. The seven steps required to solve the Towers of Hanoi problem for three disks.

machines, the stack pointer points to the top of the most recent stack frame, and on others it points to the bottom of the most recent stack frame.

When a stack organization is used, an index register or base register can be used as the stack pointer, and all memory references can be made relative to it. On the 370, a base register can be set to point to the bottom of the current stack frame, and different displacements can be used to access the various items in the current stack frame. On the Cyber 70, a B register can be used as the stack pointer, and indexing can be used to access words below the top of the stack. On the PDP-11, register 6 is generally used as the stack pointer, with indexing used to access data within the current stack frame.

Each procedure call adds a new frame to the stack, and each procedure return removes a frame from the stack. In order to illustrate the use of a stack in implementing recursive procedures, we will trace the procedure of Fig. 3-43, starting with the call

```
CALL TOWERS (3,1,3)
```

Figure 3-44(a) shows the stack just after this call has been made. The procedure first tests to see if N = 1, and upon discovering that N = 3, makes the call

```
CALL TOWERS (2,1,2)
```

```
TOWERS:  PROCEDURE(N,I,J);

/* N IS THE NUMBER OF DISKS TO BE MOVED

   I IS THE STARTING PEG
   J IS THE GOAL PEG
   6-I-J IS THE OTHER PEG (K)

   THE STATEMENT

         CALL TOWERS(4,2,3);

   CAUSES A PRINT OUT OF HOW TO MOVE 4 DISKS FROM PEG 2 TO PEG 3.

   THIS PROCEDURE SOLVES THE PROBLEM OF MOVING N DISKS FROM I TO J
   USING K AS TEMPORARY STORAGE BY BREAKING IT INTO 3 SUBPROBLEMS:

         1.  FIRST MOVE N-1 DISKS FROM I TO K
         2.  THEN MOVE 1 DISK FROM I TO J
         3.  FINALLY MOVE N-1 DISKS FROM K TO J    */

   DECLARE (N,I,J) FIXED BINARY;

   IF N = 1
      THEN PUT SKIP LIST ('MOVE A DISK FROM', I, 'TO', J);
      ELSE DO;
              CALL TOWERS(N-1,I,6-I-J);
              CALL TOWERS(1,I,J);
              CALL TOWERS(N-1,6-I-J,J);
           END;

END TOWERS;
```

Fig. 3-43. A procedure for solving the Towers of Hanoi.

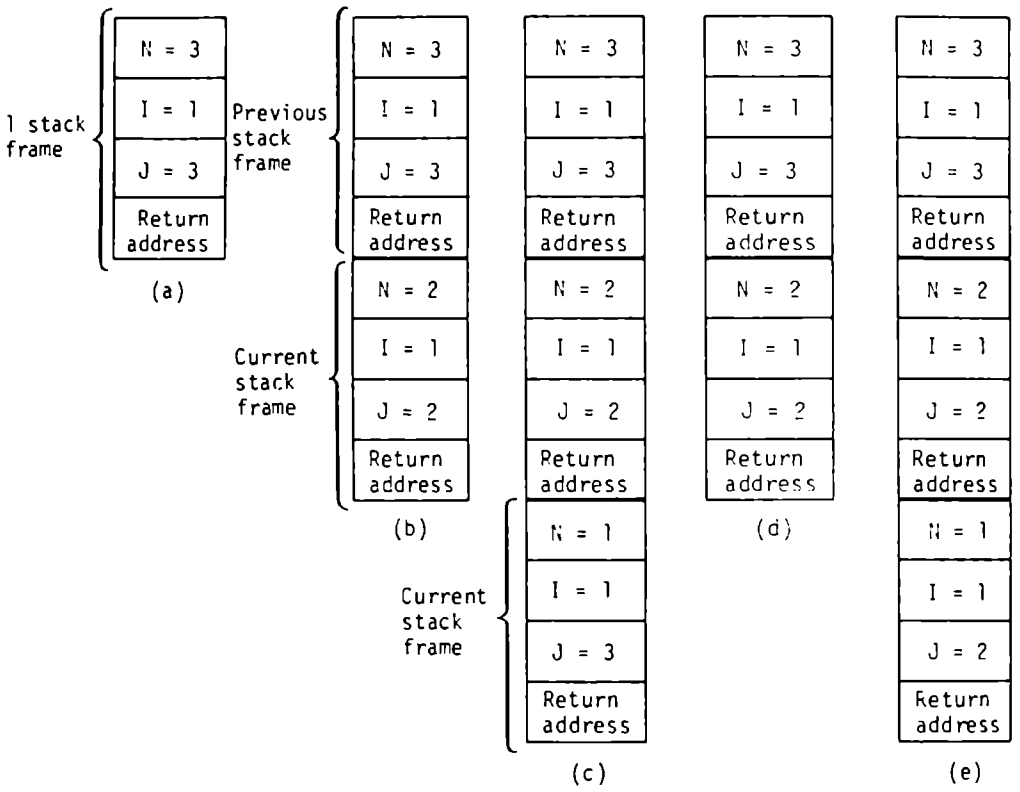


Fig. 3-44. The stack at several points during the execution of Fig. 3-43.

After this call is completed, the stack is as shown in Fig. 3-44(b), and the procedure starts again at the beginning (a called procedure always starts at the beginning). This time the test for N = 1 fails again and the call

```
CALL TOWERS (1,1,3)
```

is made. The stack then is as shown in Fig. 3-44(c), and the program counter points to the start of the procedure. This time the test succeeds and a line is printed. Next, the procedure returns by removing one frame from the stack, resetting the stack to Fig. 3-44(d). The procedure then continues executing at the return address, which is the second call:

```
CALL TOWERS (1,1,2)
```

This call adds a new frame to the stack, as shown in Fig. 3-44(e). Another line is printed; and after the return, a frame is removed from the stack. The procedure calls continue in this way until the original call completes execution, and the frame of Fig. 3-44(a) is removed from the stack.

There are several ways for a procedure to pass parameters to a procedure it calls. One way is to put them in the registers. This step is fine, providing that there are enough registers and providing that the called procedure copies the parameters to

the stack before beginning execution. A simpler method is to have the calling procedure put the parameters on the stack itself in the first place. The instructions required to call a procedure and pass parameters to it are known as the **calling sequence,** and the ones shown in Fig. 3-45 for

```
CALL TOWERS (N - 1, I, 6 - I - J)
```

are by no means the only possibilities for those machines. Because displacements on the 370 must be positive, the stack pointer must point to the bottom rather than the top of the current stack frame. The Cyber 70 does not force any choice, and we have chosen to set it to the bottom there, too. On the PDP-11, stacks grow from higher-numbered addresses to lower-numbered addresses; also, it is more efficient to have the stack pointer point to the top rather than the bottom of the stack frame. The locations of N, I, J, and the return address for the stacks of Fig. 3-45 are shown below.

	IBM 370	Cyber 70	PDP-11
N (bottom)	$k \leftarrow$ SP	$k \leftarrow$ SP	$k + 6$
I	$k + 4$	$k + 1$	$k + 4$
J	$k + 8$	$k + 2$	$k + 2$
return (top)	$k + 12$	$k + 3$	$k \leftarrow$ SP

Here k is the value of the stack pointer and SP indicates the item being pointed to.

The secret of the stack's success is a procedure's ability to automatically access a different set of parameters, local variables (if any), and a return address every time it is called, even if previous calls are not yet finished. If fixed-memory locations are used to store a procedure's parameters, local variables, and return address, the procedure cannot be called recursively.

Recursion has many applications in computer science, one of the most important being in compiler writing. A large number of compilers use recursion very heavily. For example, some compilers have a procedure for processing each construction in the language they compile, such as declarations, expressions, variables, assignment statements, and IF statements. An expression is made up of a series of terms, such as

```
A * B * C,  Y/Z,  or  R * S/T
```

separated by addition and subtraction operators. The procedure EXPRESSION may call the procedure TERM to help it. An element of a term may be a parenthesized expression, as in A + 3 * (B + C + D); therefore the procedure EXPRESSION may call the procedure TERM, which may then call EXPRESSION. This situation is called daisy chain recursion, and it is very common.

3.6.3. Coroutines

In the usual calling sequence, there is a clear distinction between the calling procedure and called procedure. Consider a procedure A, which calls a procedure B. Procedure B computes for a while and then returns to A. At first, one might consider

```
a)  IBM 370   L    1,0(6)          LOAD N INTO REGISTER 1. REG 6 IS THE STACK PTR
              S    1,=F'1'         REGISTER 1 = N - 1
              ST   1,16(6)         PUT FIRST PARAMETER INTO THE NEXT STACK FRAME
              L    2,4(6)          LOAD I INTO REGISTER 2
              ST   2,20(6)         PUT SECOND PARAMETER, I, INTO NEXT STACK FRAME
              L    1,=F'6'         REGISTER 1 = 6
              SR   1,2             REGISTER 1 = 6 - I
              S    1,8(6)          REGISTER 1 = 6 - I - J
              ST   1,24(6)         PUT THIRD PARAMETER, 6 - I - J, IN NEXT FRAME
              L    1,=A(RETURN)    REGISTER 1 = RETURN ADDRESS
              ST   1,28(6)         PUT THE RETURN ADDRESS INTO THE NEXT STACK FRAME
              A    6,=F'16'        ADVANCE THE STACK POINTER TO POINT TO NEXT FRAME
              B    TOWERS          CALL TOWERS. NOTE: RETURN ADDRESS ALREADY SAVED
    RETURN

b)  Cyber 70  SA1  B6              LOAD N INTO X1. B6 IS THE STACK POINTER
              SX6  X1-1            X6 = N - 1
              SA6  B6+4            PUT FIRST PARAMETER INTO THE NEXT STACK FRAME
              SA2  B6+1            LOAD I INTO X2
              BX6  X2              X6 = I
              SA6  B6+5            PUT SECOND PARAMETER, I, INTO THE NEXT STACK FRAME
              SX6  6               X6 = 6
              SA1  B6+2            X1 = J
              IX6  X6-X2           X6 = 6 - I
              IX6  X6-X1           X6 = 6 - I - J
              SA6  B6+6            PUT THE THIRD PARAMETER INTO THE NEXT STACK FRAME
              SX6  RETURN          X6 = RETURN ADDRESS
              SA6  B6+7            PUT THE RETURN ADDRESS INTO THE NEXT STACK FRAME
              SB6  B6+4            ADVANCE STACK POINTER
              JP   TOWERS          CALL TOWERS
    RETURN

c)  PDP-11    MOV  6(SP),-(SP)     PUSH N ONTO THE STACK. SP (R6) IS THE STACK POINTER
              DEC  (SP)            CHANGE N ON TOP OF THE STACK TO N - 1
              MOV  6(SP),R1        R1 = I
              MOV  R1,-(SP)        PUSH I ONTO THE STACK
              MOV  #6,R2           R2 = 6 (IMMEDIATE ADDRESSING)
              SUB  R1,R2           R2 = 6 - I
              SUB  6(SP),R2        R2 = 6 - I - J
              MOV  R2,-(SP)        PUSH R2 ONTO THE STACK
              JSR  PC,TOWERS       CALL TOWERS AND SAVE PROGRAM COUNTER ON THE STACK
```

Fig. 3-45. Calling sequence for CALL TOWERS (N − 1, I, 6 − I − J).
(a) IBM 370. (b) Cyber 70. (c) PDP-11.

this situation symmetric, since neither A nor B is a main program, both being procedures. (Procedure A may have been called by the main program, but that point is irrelevant.) Furthermore, first there is a transfer of control from A to B—the call—and later there is a transfer of control from B to A—the return.

The asymmetry arises from the fact that when control passes from A to B, procedure B begins executing at the beginning; when B returns to A, execution begins not at the beginning of A but at the statement following the call. If A runs for a while and calls B again, execution begins at the beginning of B again, not the statement following the previous return. If, in the course of running, A calls B many times, each and every time B starts at the beginning all over again, whereas A never starts over again, as illustrated in Fig. 3-46.

This difference is reflected in the method by which control is passed between A and B. When A calls B, it uses the procedure call instruction, which puts the return

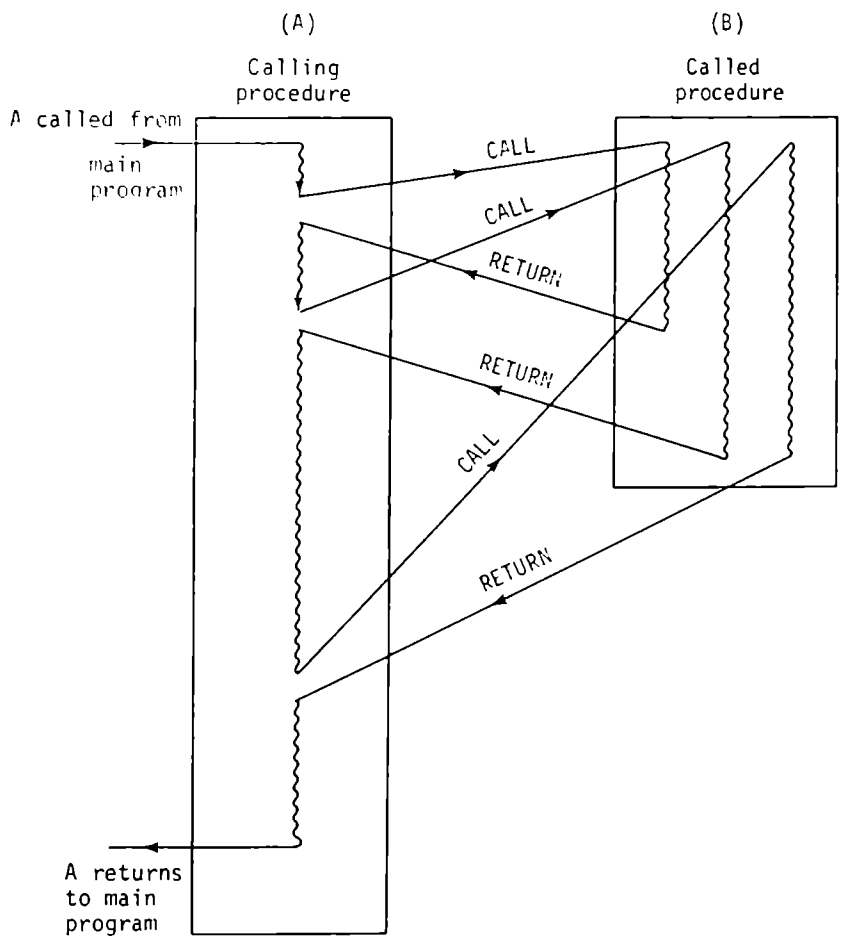


Fig. 3-46. When a procedure is called, execution of the procedure always begins at the first statement of the procedure.

address (i.e., the address of the statement following the call) somewhere useful—for example, on top of the stack. It then puts the address of B into the program counter to complete the call. When B returns, it does not use the call instruction, but instead it uses the return instruction, which simply pops the return address from the stack and puts it into the program counter.

Sometimes it is useful to have two procedures, A and B, each of which calls the other as a procedure. When B returns to A, it jumps to the statement following the call to B, as above. When A transfers control to B, it goes not to the beginning (except the first time) but to the statement following the most recent "return"—that is, the most recent call of A, as shown in Fig. 3-47. Two procedures, each of which regards the other as a procedure (in the sense that it is called, performs some work, and then returns to the statement following the call), are called **coroutines**.

Neither the usual call nor the usual return instruction will suffice to call coroutines, since the address to jump to comes from the stack like a return, but, unlike a return, the coroutine call itself puts a return address somewhere for the subsequent return to it. This operation could be done simply if there were an instruction to exchange the top of the stack with the program counter. In detail, this instruction would first pop

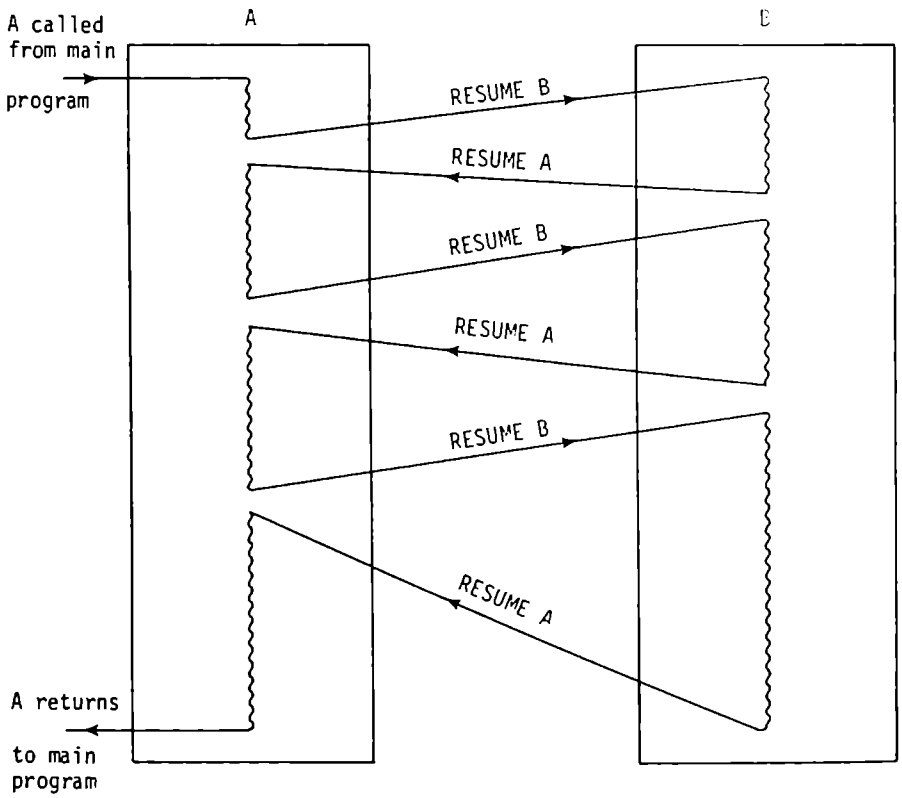


Fig. 3-47. When a coroutine is resumed, execution begins at the statement where it left off the previous time, not at the beginning.

the old return address off the stack into a temporary internal register, then push the program counter onto the stack, and, finally, transfer the contents of the temporary register into the program counter. Since one word is popped off the stack and one word is pushed onto the stack, the stack pointer does not change. A coroutine call sequence is first initialized by pushing the address of one of the coroutines onto the stack. A coroutine call instruction is sometimes called **resume**. It does, in fact, exist on some level 2 machines in the form described here. Fig. 3-48 illustrates RESUME.

The PDP-11 instruction

```
JSR PC, @(SP)+
```

performs a swap of the program counter and top of the stack. First, the mode 3, register 6 destination is evaluated. This evaluation specifies the jump address indirectly from the top of the stack and then pops the address off the stack. Next, the instruction executes, pushing the program counter onto the stack. The 370 and Cyber 70 do not have any similar instructions.

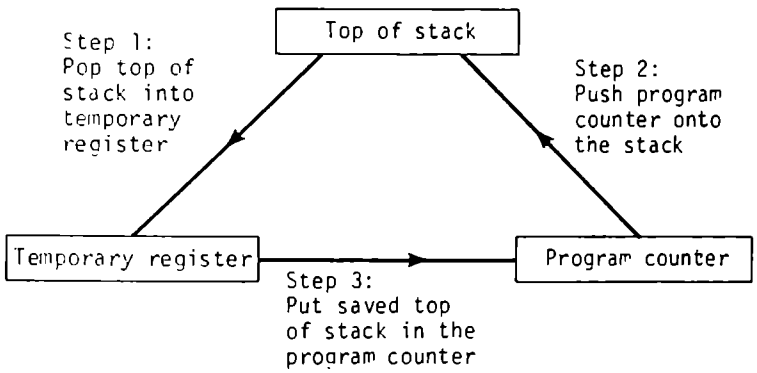


Fig. 3-48. Operation of the resume instruction.

As an example of the use of coroutines, consider a very advanced programming system that provides automatic program documentation. At any place in the program, the programmer may insert a comment preceded and followed by #. The program is used by the compiler to produce the object program. The comments are used by the documentation system to produce the manual. The compiler ignores the comments, and the documentation system ignores the program. To make the problem interesting, we will assume that the documentation part is very clever and parses all the comments to make sure that there are no grammatical errors, for programming manual writers seem to have great difficulties with the English language. A sample input is shown below.

```
IF SEATS = 350 ≒ If the plane is full, ≒
THEN DO;
    FULL = 1; ≒set a flag, and≒
    CALL NEW_PLANE; ≒request another aircraft; ≒
END;
ELSE CALL RESERVE (PASSENGER); ≒otherwise give this guy a seat. ≒
  .
  .
  .
```

The program described can be written as two coroutines—one for parsing the program and one for parsing the English. The coroutine for the program, the compiler, begins reading the input, parsing as it goes. Eventually it comes to the first comment symbol, ≒. At this point it wants to get rid of the comment so that it can continue parsing. In order to do so, the compiler calls the documentation coroutine to skip the comment.

The documentation system begins running and parsing the comment. Eventually it runs across the comment symbol. From its point of view, the program text is to be ignored, so it resumes the compiler to skip the program text. The compiler will be restarted with all its variables and internal pointers intact, from the point it left off, not the beginning. The English parser may be in a very complicated state at this point; and it is essential that when the compiler has eaten the next piece of program text, the English parser is restarted in the state it was in when it resumed the compiler.

3.6.4. Traps

A trap is a kind of automatic procedure call initiated by some condition caused by the program, usually an important but rarely occurring condition. A good example is overflow. On many computers, if the result of an arithmetic operation exceeds the largest number that can be represented, then a trap occurs, meaning that the flow of control is switched to some fixed-memory location instead of continuing in sequence. At that fixed location is a jump to a procedure called the overflow trap handler, which performs some appropriate action, such as printing an error message. If the result of an operation is within range, no trap occurs.

The essential point about a trap is that it is initiated by some exceptional condition caused by the program itself and detected by the hardware or level 1 interpreter. An alternative method of handling overflow is to have a one-bit register that is set to 1 whenever an overflow occurs. If the programmer wants to check for overflow, he must include an explicit "jump if overflow bit is set" instruction after every arithmetic instruction. Doing so would be both slow and wasteful of space. Traps save both time and memory compared with explicit programmer-controlled checking.

It is interesting to note that the trap may be implemented by an explicit test performed by the interpreter at level 1. If an overflow is detected, the trap address is loaded into the program counter. What is a trap at one level may be under program control at a lower level. Having the microprogram make the test still saves time compared to a programmer test, since it can be easily overlapped with something else.

It also saves memory, since it need only occur in a few level 1 procedures, independent of how many arithmetic instructions occur in the main program.

A few common conditions that can cause traps are floating-point overflow, floating-point underflow, integer overflow, protection violation, undefined opcode, stack overflow, attempt to start nonexistent i/o device, attempt to fetch a word from an odd-numbered address (on the 370 or PDP-11), and division by zero.

3.6.5. Interrupts

Interrupts are changes in the flow of control caused not by the running program but by something else, usually related to i/o. For example, a program may instruct the disk to start transferring information, and set the disk up to provide an interrupt as soon as the transfer is finished. Like the trap, the interrupt stops the running program and transfers control to an interrupt handler, which performs some appropriate action. When finished, the interrupt handler returns control to the interrupted program. It must restart the interrupted process in exactly the same state it was in when the interrupt occurred, which means restoring all the internal registers to their preinterrupt state.

The essential difference between traps and interrupts is that *traps* are synchronous with the program and *interrupts* are asynchronous. If the program is rerun a million times, the traps will reoccur in the same place each time, but the interrupts may vary depending, for example, on precisely when a person at a teletype pushes the carriage return key. The reason for the reproducibility of traps and irreproducibility of interrupts is that traps are caused directly by the program and interrupts are at best indirectly caused by the program.

The need for interrupts arises when input or output can proceed in parallel with CPU execution. On computers where the CPU issues an i/o instruction and then stops to wait for the i/o to be completed, there is no need for an interrupt. When the i/o is finished, the CPU is automatically restarted at the instruction following the i/o instruction. Because the CPU can generally execute many thousands of instructions during the time required to complete a single i/o instruction, it is wasteful to force the CPU to be idle during this time. An interrupt system allows the CPU to compute concurrently with the i/o and be signaled as soon as the i/o is completed.

A large computer may have many i/o devices running at the same time; for example, it might be reading data from cards, printing results on the line printer, writing output on a disk for future use, and plotting a graph of results on the plotter. All this activity can lead to complicated situations. When the card reader has finished reading a card, the CPU is interrupted and the card reader service procedure is begun. The card reader service procedure must move the card just read to the main memory location where the CPU expects it (if it is not already there), check to see if any reading errors occurred, possibly check to see if each card column contains a valid character, and issue an instruction to start reading the next card.

A nonzero probability exists that another i/o device—for example, the disk—will complete its i/o instruction before the reader service procedure has completed its

task. There are two possible ways to handle this situation. First, the disk can cause a CPU interrupt, halting execution of the card reader service procedure and starting execution of the disk service procedure. Second, the disk can be forced to wait until the reader service procedure is finished and can then cause an interrupt. We will now examine these possibilities in detail.

If we allow the disk to interrupt the card reader service procedure, we must also be prepared for the printer to interrupt the disk service procedure and for the plotter to interrupt the printer service procedure. If this interrupt sequence actually occurs, it is necessary to decide what to do when the plotter service procedure finishes. Possibilities are to continue the printer, disk, or card reader service procedures or to continue the CPU program that was running when the card reader interrupt occurred. It is clear that the administration involved in keeping track of which procedure to run *when* can get complicated.

One method for simplifying this administration is to require that all interrupts be **transparent**, which means that whenever an interrupt occurs, the state of the interrupted process is saved, including the program counter, registers, condition codes, and so on. The interrupt service procedure is then run; and, finally, the state of the interrupted process is restored to exactly the same condition it was in when the interrupt occurred and the process restarted.

The process neither requires any special precautions nor needs to be concerned with the interrupt handling. It is not even aware of its existence (unless it is timing something). Because the program running at the time of the interrupt is not aware of the fact that it has been interrupted, stopped, and later restarted, the interrupt is said to be transparent (or invisible). If all interrupts are transparent, an interrupt procedure will not even notice if it itself is interrupted.

Turning again to our earlier example, it is clear that the card reader interrupt must be transparent to the main program, the disk interrupt must be transparent to the card reader service procedure, and so forth. This implies that when the plotter service procedure finally finishes its task, the printer service procedure must be restarted, and not one of the other service procedures.

Similarly, for the printer service procedure to be transparent to the disk service procedure, the latter must be restarted from the point where it was interrupted when the printer service procedure is through. In other words, the interrupt service procedures must be restarted in the reverse order in which they occurred, as shown in Fig. 3-49. Whenever an interrupt service procedure completes its task, the most recently interrupted procedure must be restarted.

From Fig. 3-49 we see that interrupts are nested in time, meaning that a program will not be restarted until all the interrupts subsequent to it have been completely processed. Situations involving nesting are common in computer science. All nesting situations have one property in common: an inner nest is always completed before the surrounding nest is completed. A stack can often be used to implement a nesting situation. Whenever a new nest is entered, the state of the computation just before the entry is saved on the stack. Whenever a nest is exited, the state of the computation just previous to entering that nest is popped off the stack and restored.

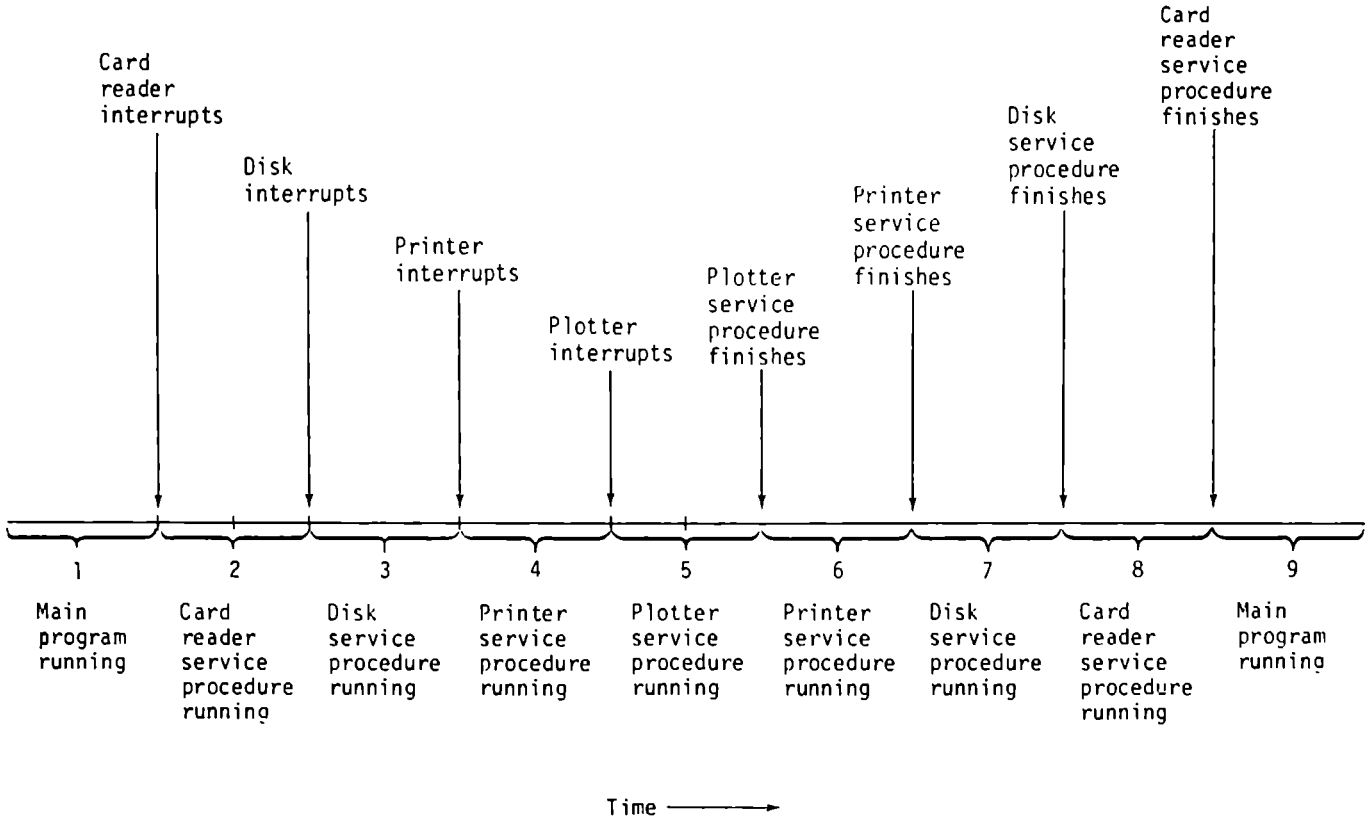


Fig. 3-49. Time sequence of interrupts from the card reader, disk, printer, and plotter.

Figure 3-50 illustrates the use of a stack for processing interrupts. The numbers 1 to 9 represent the time intervals shown in Fig. 3-49. During interval 1, no preceding state need be remembered, and the stack is empty. After the card reader interrupt has occurred, the state of the main program at the time of the interrupt must be remembered so that it can be restarted in the correct place later. This situation is shown as 2. When the disk interrupts the card reader service procedure, the state of the card reader service procedure must also be stacked, shown as 3.

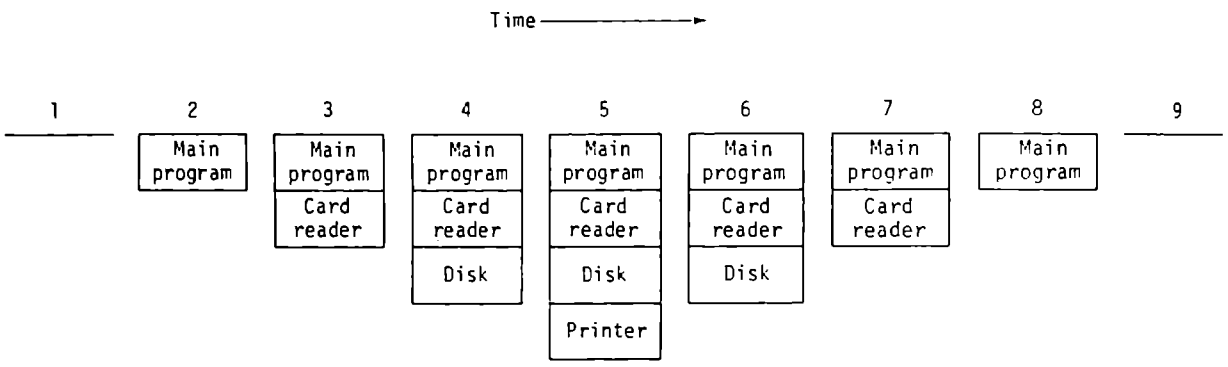


Fig. 3-50. Use of a stack to save the state of the running process when an interrupt occurs. Each box is the saved state of the indicated process. The numbers correspond to the time intervals of Fig. 3-49.

Each of the nine stack configurations refers to some sequence of as-yet-uncompleted interrupt procedures. If another interrupt occurs at 7, while the disk service procedure is running, the state of that procedure will be saved again. If the computer posesses many i/o devices, a given interrupt service procedure may be stopped and resumed several times before it completes.

The model we have just given for interrupt processing is, however, not complete because we have ignored the critical timing aspects of i/o devices. Some i/o devices must be serviced within a specific time interval or information will be lost. For example, if data is being transmitted over a communications line at 960 characters/sec, a character arrives in the receiver buffer every 1042 μsec. If the interrupt service procedure fails to fetch the character within 1042 μsec, it may be overwritten by the next one and lost. An interrupt system needs a provision for handling this kind of problem. In other words, once the service procedure for a highly critical interrupt has begun, it must not be interrupted by a less critical interrupt.

When an i/o processor on the 370 finishes executing its program, it can interrupt the CPU. It does so in the following steps. The CPU has a 64-bit register called a **program status word** (PSW) that contains the program counter, condition code, and other pieces of status information. The PSW is stored in location 56, and a new PSW is loaded from location 120. As soon as the new PSW has been loaded, the CPU

begins execution at the start of a general interrupt service procedure. The interrupt service procedure must first determine which i/o device finished by examining information stored in memory by the i/o processor. Then it can call the appropriate service procedure.

If a second interrupt occurred while the first one was being processed, the current PSW would be stored at location 56, thereby erasing the one already there. The situation is similar to a procedure call instruction that always puts the return address in a fixed place in memory. To prevent having a PSW overwritten before the interrupt procedure has had a chance to save it, the 370 has a bit associated with each i/o processor called a **mask bit**. When this bit is a 0, interrupts from that i/o processor are forced to wait until it becomes a 1. Setting a mask bit to 0 is called **disabling** an i/o processor.

The Cyber 70 CPU can only be interrupted by the PPUs, and it has no way of disabling them. When a PPU interrupts the CPU, the entire state of the CPU process, including all the registers, is saved, and a new set of registers is loaded from a memory address specified by the interrupting PPU.

When the PDP-11 is interrupted by an i/o device, the PSW (Processor Status Word) and program counter are pushed onto the stack, and a new PSW and program counter are loaded from the memory address associated with the i/o device. These memory addresses are called **interrupt vectors**, and each device has a unique one. Each interrupt vector contains the starting address of the service procedure for the corresponding device, thus eliminating the need to test which device caused the interrupt. The 370 has only one interrupt vector for all i/o devices, so the interrupt handler must first determine which device wants attention and only then can it call the proper service procedure.

The PDP-11 has a system of priority interrupts. Each i/o device has a priority number associated with it. The CPU also has a priority (from 0 to 7), which can be set by the program. If the priority of the i/o device is higher than the current CPU priority, the interrupt takes place; otherwise it is forced to wait until the CPU priority is reduced. This feature can be used to ensure that time-critical interrupt service procedures can only be interrupted by still more critical ones, not by less critical ones. For example, if the disk interrupt service procedure runs at priority 5, an interrupt from magnetic tape at priority 6 can interrupt it, but an interrupt from the paper tape reader at priority 4 will be forced to wait until the CPU priority is set to 3 or less, which will not occur until the critical disk service procedure is finished. The priority of an i/o device can be set by a simple adjustment to the device itself. The devices that need the fastest service are naturally given the highest priorities.

PROBLEMS

1. Describe how the following are used:
   a. expanding opcodes
   b. indexing

c. base registers
d. stack arithmetic
e. self-relative addressing
f. rotate instructions
g. dope vectors
h. interrupts

2. Why do computers often have eight or more registers? In what way do they complicate programming? Can you envision any technological developments that will remove their reason for existence?

3. Design an expanding opcode to allow all the following to be encoded in a 36-bit instruction:
   7 instructions with two 15-bit addresses and one 3-bit address
   500 instructions with one 15-bit address and one 3-bit address
   50 instructions with zero addresses

4. Is it possible to design an expanding opcode to allow the following to be encoded in a 12-bit instruction? An address is three bits.
   4 instructions with three addresses
   255 instructions with one address
   16 instructions with zero addresses

5. A certain machine has 16-bit instructions and 6-bit addresses. Some instructions have one address and others have two addresses. If there are N two-address instructions, what is the maximum number of one-address instructions?

6. Given the memory values below, and a one-address machine with an accumulator, what values do the following instructions load into the accumulator?
   word 20 contains 40
   word 30 contains 50
   word 40 contains 60
   word 50 contains 70
   a. LOAD IMMEDIATE 20
   b. LOAD DIRECT 20
   c. LOAD INDIRECT 20
   d. LOAD IMMEDIATE 30
   e. LOAD DIRECT 30
   f. LOAD INDIRECT 30

7. Compare 0, 1, 2, and 3-address machines by writing programs to compute
   X = (A + B × C)/(D − E × F − G × H)
   for each of the four types of instruction sets. The available instructions are:

0 Address	1 Address	2 Address	3 Address
PUSH M	LOAD M	MOV (X = Y)	MOV (X = Y)
POP M	STORE M	ADD (X = X + Y)	ADD (X = Y + Z)
ADD	ADD M	SUB (X = X − Y)	SUB (X = Y − Z)
SUB	SUB M	MUL (X = X * Y)	MUL (X = Y * Z)
MUL	MUL M	DIV (X = X/Y)	DIV (X = Y/Z)
DIV	DIV M		

where M is a 16-bit memory address and X, Y, and Z are either 16-bit memory addresses or 4-bit register numbers. The 0-address machine uses a stack, the 1-address machine uses an accumulator, and the other two have 16 registers and instructions in which either operand may be in a register or in memory. Assuming eight-bit opcodes and instruction lengths that are a multiple of four bits, how many bits does each computer need to calculate X?

8. Write a conventional machine level program for some computer to compute the inner product of two N-element vectors, using indexing.

   inner product = A(1) × B(1) + A(2) × B(2) + A(3) × B(3) + . . . + A(N) × B(N)

9. Repeat the calculation of Problem 8 but without using indexing this time.

10. Base register addressing involves three lengths:

    L1 = length of the field specifying a base register

    L2 = length of the field specifying a displacement

    L3 = length of a base register

    a. How many distinct addresses can be referenced without changing base registers?
    b. If memory addresses require L4 bits to be specified, what fraction of the total memory can be accessed without changing base registers?

11. Devise an addressing mechanism that allows an arbitrary 64 addresses in a large address space to be specifiable in a six-bit field.

12. An alternative design for base register machines like the 370 is to provide 24 bits in each instruction for a direct address instead of a base register field and a displacement field, which together require only 16 bits. Evaluate these alternatives for the following cases:
    a. The data consists of a few clumps scattered throughout the address space.
    b. The data is uniformly distributed throughout the address space.
    Which of these cases did the 370 designers think was more probable?

13. Convert the following formulas from infix to reverse Polish:
    a. A+B+C+D+E
    b. A×B×C×D×E
    c. (A+B)×(C+D)×E
    d. (A×B)+(C×D)+(E×F)
    e. ((A−B)×((A−B×G)/E)/F)×H

14. Convert the following reverse Polish formulas to infix:
    a. AB+C+D×
    b. AB/CD/×
    c. ABCDE+///
    d. AB+C−D−E+AB+C−D−E/×
    e. GFABC×E/+F−E/×+

15. Which of the following pairs of reverse Polish formulas are mathematically equivalent?
    a. AB+C+ and ABC++
    b. AB−C− and ABC−−
    c. AB×C+ and ABC+×

16. Write three reverse Polish formulas that cannot be converted to infix—that is, which are invalid.

17. Convert the following infix boolean formulas to reverse Polish:

    a. $(A \wedge B) \vee C$
    b. $(A \vee B) \wedge (A \vee B)$
    c. $(A \wedge B) \vee (A \wedge B)$
    d. $(A \vee B) \wedge (A \wedge B) \wedge (C \vee D)$

18. Convert the following infix formula to reverse Polish and evaluate it, using a stack. Show each step as in Fig. 3-29.

    $(2*3+4)-(4/2+1)$

19. Convert the following infix formula to reverse Polish and evaluate it, using a stack. Show each step as in Fig. 3-29.

    $(3\times(4+(6\times(5+(7\times(8+4))))))\times(((2-4)-3)+6)$

20. Your company, like many others before it, is planning to launch a line of computers compatible with the IBM 370. To help sales, it has been decided that your machine must be more powerful than the 370 in some way, but it also must be able to run existing 370 programs. One group in your company is blindly in love with indirect addressing and wants to produce a machine (to be called the 371) that allows indirect addressing on all RX instructions. Another group wants to produce a machine (to be called the 370.01) that has two additional instructions not found on the 370: load register indirectly from memory and store register indirectly into memory. Is either proposal feasible? Explain.

21. The company of Problem 20 went bankrupt, and you found a new job with a company that wants to produce a PDP-11 with a 32-bit word—that is, four bytes instead of two. Is such a design possible without greatly altering the entire PDP-11 addressing structure and architecture? (This machine need not run existing PDP-11 programs.) Discuss the design.

22. It is very common in programming for a program to need to determine where a variable X is with respect to the interval A to B. If a three-address instruction were available with addresses specifying X, A, and B, how could you record the result, using conditions codes? How many cases are there?

23. One of your friends has just burst into your room at 2 A.M., out of breath, to tell you about his brilliant new idea: an instruction with two opcodes. Should you send him off to the patent office or back to the drawing board?

24. Tests of the form

    ```
    IF A = 0 THEN ... ELSE ...
    IF K < N THEN ... ELSE ...
    IF M >= 4 THEN ... ELSE ...
    ```

    are very common in programming. Devise an instruction to perform these tests efficiently. What fields are present in your instruction?

25. For some computer with which you are familiar, give an example of
    a. a data movement instruction.
    b. a boolean instruction.

c. an integer arithmetic instruction.
d. a shift instruction.
e. a conditional jump instruction.
f. a procedure call instruction.
g. a loop control instruction.

26. Design an orthogonal MOVE instruction that can move 1, 2, 4, or 8 bytes from the source to the destination, using the following list of sources and destinations (state any assumptions you make—for example, number of registers):

directly addressed memory
stack (as a source this is a pop; as a destination this is a push)
indirectly addresses memory
immediate
general register
floating-point register
indexed
indexed indirect

How many combinations are there? Are they all useful?

27. For the 16-bit binary number 1001 0101 1100 0001, show the effect of
a. right shift of 4 bits with zero fill.
b. right shift of 4 bits with sign extension.
c. left shift of 4 bits.
d. left rotate of 4 bits.
e. right rotate of 4 bits.

28. How can you clear a memory word to 0 on a machine with no clear word instruction?

29. Compute the boolean expression (A AND B) OR C for

```
A = 1101  0000  1010  1101
B = 0000  0000  1111  0000
C = 0000  0000  0000  0000
```

30. Devise a sequence of instructions to interchange two variables A and B without using a third variable or register, using only the instruction EXCLUSIVE OR.

31. On a certain computer it is possible to move a number from one register to another, shift each of them left by an arbitrary amount, and add the shifted registers in less time than it takes to do a multiplication. What condition must the constant satisfy for the shifting and adding algorithm to be able to compute "constant times variable" faster than a multiplication?

32. The Cyber 70 is a 1's complement machine that has neither a compare instruction for integers nor an overflow indicator. A way of computing

```
IF I < J THEN ... ELSE ...
```

is to compute I − J and then jump to the ELSE part if the sign bit of the result is 0. Give two different situations where this algorithm fails. Give a Cyber 70 program that works for all integers less than 2^{60}.

33. Design an instruction to be used for test-at-the-beginning loops.

34. To save space, one-dimensional boolean arrays are packed N bits per word. Write a procedure in assembly language that takes as parameters the address of the first word of booleans and the element required (the numbering starts at 0) and produces as a result an integer that is 0 if the boolean is 0 and nonzero otherwise.

35. Show the tables needed for marginal indexing of an array A (1:4, 1:2, 1:4).

36. Consider an array A (0:3, 2:7). Write down the expression giving the address of A(I, J) using the dope vector system, if the elements are stored one per address beginning at 1000.

37. What are the advantages and disadvantages of packing character strings several per word, as compared to using a linked list?

38. Write an assembly language procedure to compute N!, using iteration.

39. Write an assembly language procedure to compute N!, using recursion.

40. In what situations would the parameter mechanism used in Fig. 3-45 be inadequate? Propose a better one.

41. Devise a sequence of two or more instructions to resume a coroutine. Assume that the return address is on the stack. Specify where the stack pointer is.

42. What is the essential difference between a trap and an interrupt?

43. The PDP-11 and some other computers have a system of priority interrupts. Is it a reasonable idea also to have a system of priority traps? Why or why not?

44. If you are not convinced that recursion is, at times, an indispensable programming technique, try to write a procedure in any language to solve the "Towers of Hanoi" problem without using recursion. Be warned, however, that you will probably not be able to find the solution.

45. On a machine using base registers, it is necessary for an interrupt handler to save one base register for use by the interrupt procedure itself. However, the instruction to store a base register itself requires the use of a base register to address the word where the base register is to be stored. Because an interrupt can occur at any time, the values of all the base registers are unknown at the start of the interrupt handler. This means it has no base register available to use for the store base register instruction. Devise a solution to this problem. If you are using a 370 or other base register machine, check to see how the manufacturer solved this problem.

47. Write a procedure that simulates the addition of floating-point numbers, using the single-precision format used by your computer. Floating-point numbers are represented as integer arrays, with each word containing either a 1 or a 0. An N-bit floating-point number will require an N-word array.

48. How could Huffman coding be employed to shorten the instruction length for a machine in which an instruction could be any number of bits?

4 THE MICROPROGRAMMING LEVEL

The boundary between hardware and software is not well defined and, furthermore, is constantly shifting. Early computers had instruction sets and organization similar to those described in the preceding chapter. On these early machines, the instructions for arithmetic, boolean operations, shifting, comparing, looping, and so on were all directly executed by the hardware. In other words, there were specific circuits for each instruction. One could unscrew the back panel and point to the electronic components used by the division instruction, at least in principle.

On a modern multilevel computer, it is no longer possible to isolate the division circuits because there are no division circuits. All the instructions available at the conventional machine level (the arithmetic, boolean, shifting, comparing, looping, etc. instructions) are carried out one step at a time by an interpreter running at the microprogramming level. The modern-day equivalent of looking for the division circuits is to get out a listing of the microprogram and look for that portion of it which interprets division instructions.

Although programs at any level may be carried out by a software interpreter, and although this interpreter itself may also be executed by another interpreter, this hierarchy cannot go on indefinitely. At the very bottom level there must be a physical hardware machine, with integrated circuits, power supplies, and similar "hard" objects. It is the purpose of this chapter to study the organization of these machines, the level they constitute, which we call the microprogramming level, and their programming, called microprogramming.

Because the microprogramming level is defined by the actual hardware, it is usually very primitive and awkward to program. Timing considerations are often

important, for example. Consequently, Rosin (1974) has defined microprogramming as "the implementation of hopefully reasonable systems through interpretation on unreasonable machines."

The microprogramming level has a specific function: to execute interpreters for other (hopefully, more reasonable) virtual machines. This design goal naturally leads to an organization highly optimized toward fetching, examining, and executing conventional machine instructions, and, in some cases, more sophisticated instructions. The issues and tradeoffs involved in the design of the organization and instructions for this level will be examined in this chapter.

We will begin our study of the microprogramming level by briefly examining the basic building blocks from which hardware processors are constructed. Our concern is how they appear to the microprogrammer rather than how they work inside. Next, we come to the heart of the subject, how more complex instructions can be built up from sequences of very simple instructions. This topic will be discussed in detail and illustrated by a simple hypothetical two-level computer. Afterward we will examine the various factors that must be considered when designing a computer's microprogramming level in order to understand better why it is the way it is. Finally, we will examine the microprogramming level of some computers that have actually been produced and sold commercially.

4.1. PROCESSOR COMPONENTS

In the following sections, the components making up the CPU will be examined from the standpoint of the microprogrammer. These components each have specialized functions, including storing, transferring, and controlling the flow of information. Their number, type, and interconnection form the architecture of the microprogramming level. Although we will describe these components as separate entities, they may be simply different, specialized portions of the same integrated circuit chip.

4.1.1. Registers

A register is a device capable of storing information. The microprogramming level always has some registers available to hold information needed for the processing of the instruction currently being interpreted. Conceptually, registers are the same as main memory, the difference being that the registers are physically located within the processor itself, so they can be read from and stored into faster than words in main memory. In the future, computers are likely to be constructed with integrated circuit memory physically housed within the same chip as the processor. Larger and more expensive machines usually have more registers than smaller and cheaper ones, which have to use main memory for storing intermediate results. On some computers a group of registers numbered 0, 1, 2, . . . , $N - 1$, is available at the microprogramming level, and is called **local storage** or **scratchpad storage**.

A register can be characterized by a single number: how many bits it can hold. Figure 4-1 shows a 16-bit register with the bit-numbering convention used in this

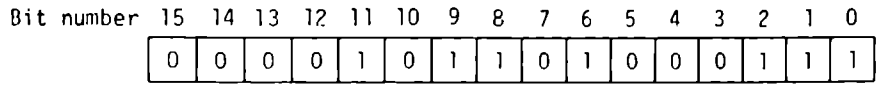


Fig. 4-1. A 16-bit register is a device for holding 16 bits of information.

book. Information placed in a register remains there until some other information replaces it. The process of reading information out of a register does not affect the contents of the register. In other words, when a register is read, a copy is made of its contents and the original is left undisturbed in the register.

4.1.2. Buses

The electrical connection between two registers is called a **bus**. A bus consists of a number of parallel wires, one for each bit in the registers. Two 16-bit registers will be connected by a 16-wire bus. The width of such a bus is said to be 16. In fact, a bus normally contains a few extra wires for synchronization and control information, but that fact will not be of concern to us.

Bits are transferred along the wires of the bus in parallel. At the same time as bit 0 is being sent along its wire, bit 1 is speeding along its wire, and so on. The reason that registers are connected by multiwire buses instead of by a single wire is to allow information to be transferred in parallel, both because it is simpler and because it is faster. When drawing diagrams of the microprogramming level organization, we will draw buses as single lines, just to keep the diagrams from getting too cluttered. Information transfer between registers will always be in parallel.

Although a wire permits signals to travel in either direction, for our purpose, a given bus will only be used to transmit information in one direction, indicated by an arrow on the bus. The one-way property is caused by the circuits connecting the bus to the source and destination registers, not by any property of the bus itself.

4.1.3. Gates

AND gates are electronic switches that control the flow of information out of registers onto buses and from buses into registers. Figure 4-2 illustrates the use of a gate. A gate has two inputs and one output. One input is the data signal and the other is the control signal. If the control signal is 1, the data passes through the gate unimpeded, as if no gate were present at all. If the control signal is 0, no information can pass through the gate. The data signal can, and usually does, consist of a number of bits transferred in parallel.

An analogy can be made between a gate and a traffic light on a multilane one-way street. When the light is red (control signal is 0), no cars appear on the downstream side of the light in any of the lanes (output signal is 0). A car will appear in a particular lane on the downstream side of the light (output signal is 1) if and only if a car appears on the upstream side of the light (data signal is 1) and the light is green (control signal is 1).

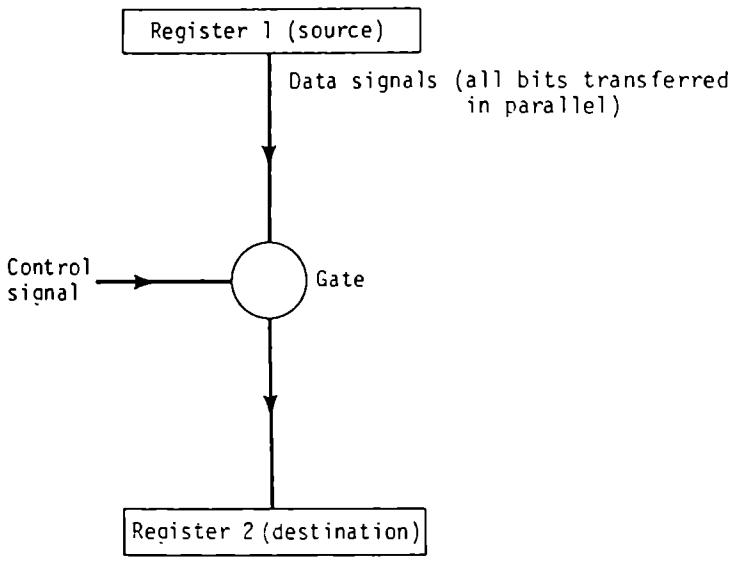


Fig. 4-2. A gate can control the flow of information between two registers. When the control signal is 1, the source register is copied into the destination register. When the control signal is 0, the destination register remains unchanged.

A gate performs the boolean AND function: if the control signal and the data signal are both 1, the output is 1; if either the control signal or the data signal is 0, or if both are 0, the output is 0. There also exist OR gates, as well as other kinds of gates, but we will only be concerned with AND gates, and the term "gate" will be used to mean an AND gate.

The data signal entering a gate will generally be a bus containing more than one wire. The output contains as many wires as the input. The output signal on each wire is the boolean AND of its input and the control signal. Or, stated more simply, if the control signal is 1, the output signal on the wire is the same as the input signal. If the control signal is 0, the output signal on the wire is 0, independent of the input signal.

4.1.4. Clocks

Most computers contain a clock that emits pulses at a constant rate. These pulses are used to synchronize the actions going on within the processor. The interval between two clock pulses is called a **cycle** (or a processor cycle). On some machines, there are one or more smaller pulses within a cycle, dividing the cycle into subcycles. Clocks are used to break time up into discrete chunks. Figure 4-3 illustrates a clock that emits a major pulse every 30 nsec and a minor pulse every 10 nsec. (A nanosecond is 10^{-9} seconds.)

The purpose of imposing a discrete time scale on a computer is to make it easier to specify when certain actions are to occur. For example, if a microprogram using

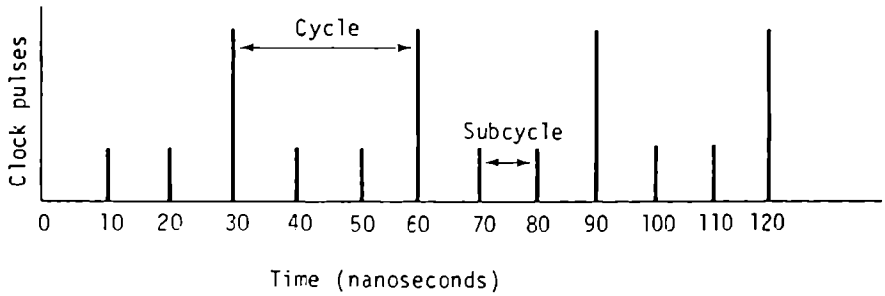


Fig. 4-3. Two different pulse heights can be used to divide time into discrete cycles and subcycles.

the clock of Fig. 4-3 determines that a certain gate must be opened (control signal set to 1), starting in 20 nsec and lasting for 10 nsec, it can wait for two small clock pulses and then open the gate until the first following clock pulse.

4.1.5. Memory Ports

Associated with each memory accessible on the microprogramming level are two registers: the memory address register (MAR) and the memory buffer register (MBR). When the address of a memory location is put into the MAR and a read signal sent to the memory, the contents of that location is loaded into the MBR. Similarly, the contents of the MBR is written into the memory at the address specified by the MAR when a write signal is given to the memory. The time it takes for the memory to accept a read signal, load the MBR with the contents of the location specified by the MAR, and get ready to accept the next read signal is called the **memory cycle** time. The memory cycle time may be different from the processor cycle time.

The number of bits in the MBR, called the **memory bus width**, plays an important role in the design of the microprogramming level. The more bits the MBR contains, the fewer memory cycles are needed to fetch a given amount of information and the faster the computer can run (assuming that memory access is the limiting factor). Of course, the more bits the MBR contains, the more expensive it is, so there is a tradeoff of performance against cost. In computer families like the 370, the slower, cheaper models have fewer bits in their MBRs than the faster, more expensive models.

Another technique is available for increasing the effective memory transfer rate without resorting to wider memory buses. This method consists of having several identical but separate memories, which can operate in parallel. Figure 4-4 gives an example of how four memories of 2^{14} words each could be used to implement an address space consisting of 2^{16} words. When the microprogram needs to fetch a new word (e.g., the next conventional machine level instruction to be interpreted), it divides the 16-bit conventional machine level address into two parts. The rightmost two bits specify the memory to be used, 0, 1, 2, or 3, and the other 14 bits specify a word within that memory.

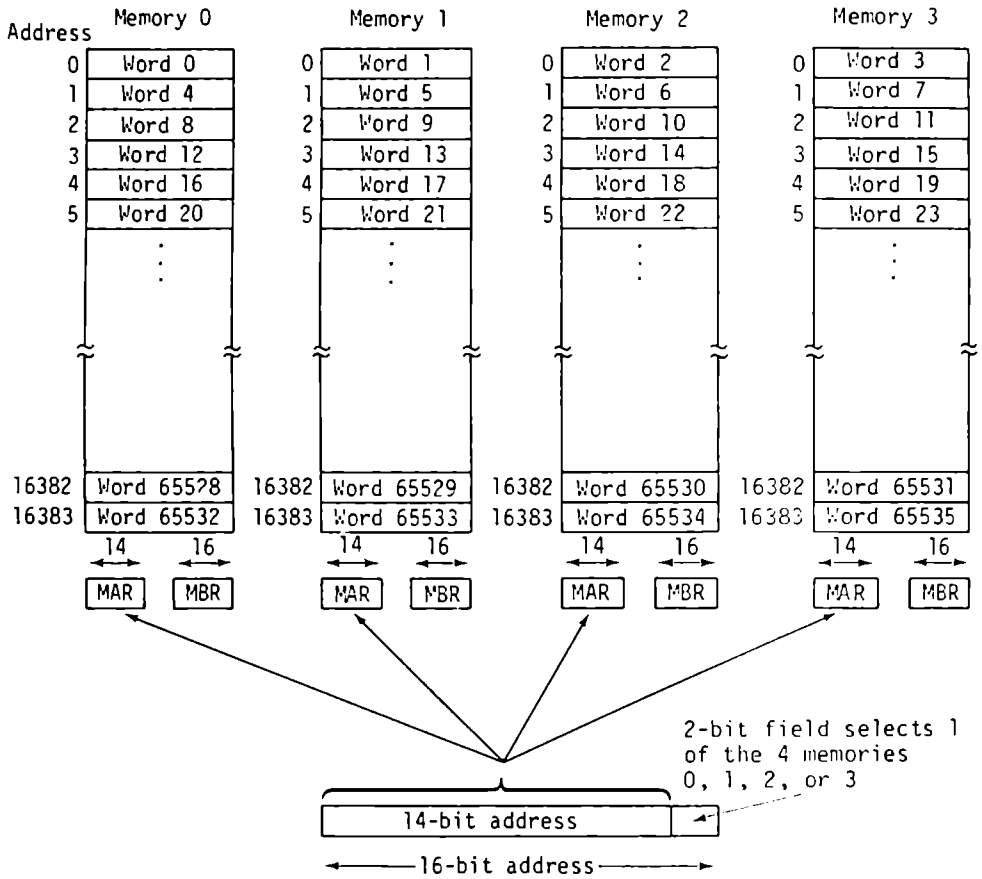


Fig. 4-4. Four distinct memories of 2^{14} words each can be used to implement a single 2^{16} word address space.

After the microprogram has determined which memory to use, it puts the 14 leftmost bits of the address in the selected MAR and starts that memory reading. While waiting for that memory to finish, the microprogram could use one of the other memories to fetch another word. Note that consecutive words at addresses K, K + 1, K + 2, and K + 3 are always located in different memories. As a result, up to four words (and four instructions, if each word contains one instruction) can be fetched in parallel. Note also that this method is independent of the MBR width. The term N-way interleaving is sometimes used to describe an organization of this kind involving N memories.

4.1.6. Arithmetic and Logical Units

Another basic processor component is the arithmetic and logical unit, or ALU. This unit takes two registers as input and computes some function of them as output. ALUs vary from machine to machine, but nearly all of them can perform addi-

tion on binary numbers, and often subtraction, too. Boolean operations such as AND are also common.

The criteria by which a computer designer must decide what his ALU should do are basically two: what is required in a strictly mathematical sense, and what is required to provide the necessary performance within the allowed budget.

As an example of the first criterion, it is sufficient to have an adder and a 1's complementor in order to provide subtraction, multiplication, and division. Subtraction is not strictly necessary because it can be accomplished by addition of the negative of the minuend. Multiplication can be carried out by repeated addition. After all, 420 times 206 is simply 420 added up 206 times. Similarly, division is merely repeated subtraction.

Nevertheless, in order to achieve a certain level of performance, it may be desirable to use an ALU with the capability for subtraction, multiplication, and division built in. A hardware multiply unit can usually outperform a program doing multiplications in software, even if they use the same algorithm. Floating-point arithmetic is another example where the choice between hard-wired and explicitly programmed operations must be determined on the basis of performance and cost.

Boolean operations are another case in point. If the ALU can perform NAND (see Appendix C), the boolean functions AND, OR, EXCLUSIVE OR, and EQUIVALENCE can be built up from repeated use of it, according to the formulas given in Fig. 4-5. Nevertheless, it may be worth the cost to provide these functions in the ALU hardware in order to gain speed.

4.1.7. Packaging of Processor Components

Although, conceptually, each bit in each register and each gate is a separate entity, neither bits, nor registers, nor gates are manufactured or sold individually. A single

P	Q	AND $P \wedge Q$	OR $P \vee Q$	EXCLUSIVE OR $P \veebar Q$	EQUIVALENCE $P \equiv Q$	NAND $P \mid Q$
0	0	0	0	0	1	1
0	1	0	1	1	0	1
1	0	0	1	1	0	1
1	1	1	1	0	1	0

(a)

$P \wedge Q = (P \mid Q) \mid (P \mid Q)$
$P \vee Q = (P \mid P) \mid (Q \mid Q)$
$P \veebar Q = ((P \mid P) \mid Q) \mid (P \mid (Q \mid Q))$
$P \equiv Q = (P \mid Q) \mid ((P \mid P) \mid (Q \mid Q))$

(b)

Fig. 4-5. (a) Definition of the boolean functions AND, OR, EXCLUSIVE OR, EQUIVALENCE, and NAND. (b) Construction of these functions by repeated use of NAND.

integrated-circuit chip may contain many distinct registers and gates, packaged together as a single unit in order to simplify construction and reduce the manufacturing cost. Similarly, an ALU might be a single, inexpensive chip or part of a chip, capable of performing addition, subtraction, and all 16 boolean functions of two variables on 16- (or 32-) bit numbers.

Throughout this chapter the microprogramming level will be described in terms of registers and gates as though they were unrelated units soldered together, because, in principle, such is the way they are used. Nevertheless, the physical organization of the components may be very different than the logical one. By standardizing on a relatively small number of integrated circuit types, manufacturers can reduce costs by mass production.

4.2. BASIC OPERATIONS

The microprogramming level has a few basic operations from which all the conventional machine level instructions can be built. These basic operations are described in the following sections. The microprogramming level's instructions are called **microinstructions.**

4.2.1. Register Transfer

The most basic of all operations is the transfer of information from one register to another. Two registers are connected by means of a bus with a gate in it. We will consider only one-way buses; that is, if a bus is used to transfer information from register A to register B, it cannot also be used to transfer information from register B to register A. If information transfer in both directions is required, a second bus is needed. A bus between two elements in a computer system used to transmit data is also called a data path and is characterized by its width, the number of bits that can be transferred in parallel.

Register transfers can be classified as direct, skew, scatter, and collect, as illustrated in Fig. 4-6. In a direct transfer, an N-bit register is copied into an N-bit register with the ith bit in the copy being the ith bit in the original. Most transfers that we will be concerned with are of this type.

A skew transfer is one in which bit i in the copy comes from bit $i + k$ in the original, k being positive or negative. In a skew transfer, one or more bits at one end of the destination have no source. Such bits are filled with 0. A scatter transfer is one in which the contents of a register is scattered among several registers, possibly with the bits in a different order. A collect transfer is one in which contents of a register is obtained from several sources, possibly with bits in a different order. Bits that have no source are set to 0.

A register transfer changes the contents of the destination register(s) but does not change the source. In Fig. 4-6(a), after the transfer, both R1 and R2 have the same value.

Figure 4-7 shows how a shift unit works. The number to be shifted left one bit

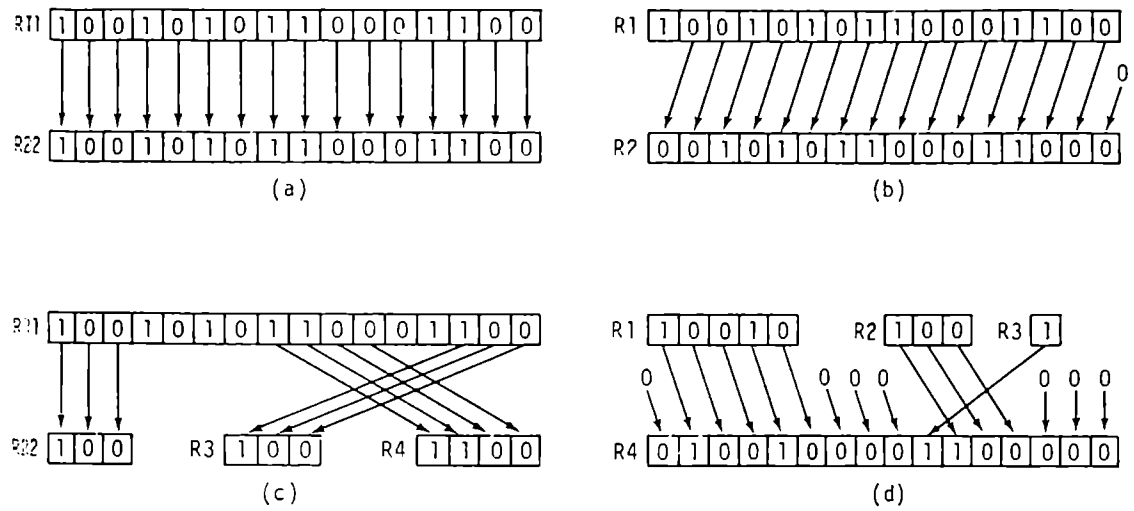


Fig. 4-6. Register transfers. (a) Direct transfer. (b) Skew transfer. (c) Scatter transfer. (d) Collect transfer.

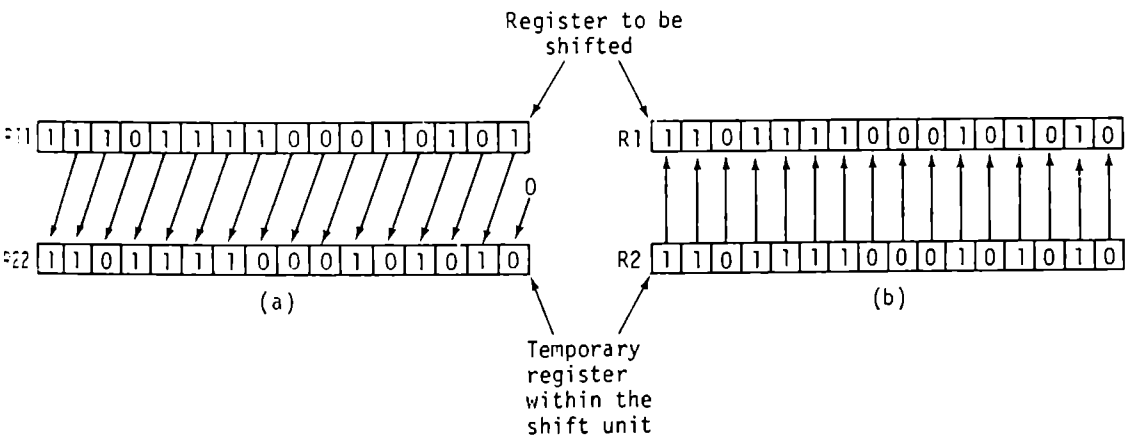


Fig. 4-7. Operation of a 1-bit left shift unit. (a) Step 1 is a skew transfer to a register internal to the shift unit. (b) Step 2 is a direct transfer back to the original register.

is placed in the shift register. A skew transfer is performed into an internal register. A direct transfer is then performed from the internal register to the shift register, along a different data path. To shift a register N bits left, this process is repeated N times. If the shifter is to be used for right shifts as well as left shifts, another data path is needed from shift register to the internal register, as in Fig. 4-7(a), but slanting to the right instead of to the left. It is noteworthy that even the internal operation of a true hardware unit, such as a shifter, is given (however, informally) as an algorithm.

4.2.2. Memory Read/Write

An operation to initiate a transfer between the memory and the memory buffer register (MBR) is necessary. It is important to remember that two kinds of transfers exist: a read, in which the contents of the cell or cells beginning at the address in the MAR is loaded into the MBR; and a write, in which the contents of the MBR is written into the memory cell or cells addressed by the MAR.

One possible mechanism is an explicit command for reading and another one for writing. A second mechanism would be to initiate a memory reference whenever a new address was gated into the MAR, with the status of a certain bit or gate determining whether a read or a write is needed.

4.2.3. Bit Testing

A computer program with no way of testing its data and then branching accordingly would carry out the same steps no matter what the data was. Such a program might possibly be adequate for some applications but certainly would not be adequate in general. A microprogram written to fetch an instruction from main memory and execute it certainly has to be able to tell a STORE instruction from a HALT instruction and be able to proceed appropriately. As a bare minimum, there must be a way to test one bit in some register and alter the flow sequence of control, depending on the outcome of the test.

One control mechanism is to provide the microprogramming level with an explicit instruction to test the bits of a specified register. An alternative method is for one or more fields within the microinstruction itself to, together with the value of one or more registers, determine the address of the next microinstruction to be executed. Both methods are widely used.

4.3. A HYPOTHETICAL TARGET LEVEL

In discussing the interpretation of one (virtual) machine by another, it is useful to have a consistent terminology in order to avoid confusing them. The level whose programs are being executed interpretively is called the **target** level. The level on which the interpreter runs is called the **host** level. Similarly, the target machine's instruction set is the one that the interpreter has to analyze and execute. The interpreter itself is written using the host level's instruction set. In this chapter the target level is the conventional machine level and the host level is the microprogramming level. The host and target levels are always different levels of the same computer. We will use the terms "host machine" and "host level" (as well as "target machine" and "target level") interchangeably throughout the book. Remember that a machine may be a virtual machine.

In order to illustrate the ideas of microprogramming, we will now describe a target machine (that is, the architecture of a conventional machine level) to be used repeatedly in subsequent examples. This target machine is a minicomputer with a

16-bit word. Addresses 0, 1, 2, etc. will refer to consecutive words (unlike the PDP-11, in which consecutive addresses refer to consecutive bytes). Memory will consist of 2^{13} words of 16 bits each, which means that addresses will have 13 bits. The MAR has 13 bits, in order to contain an address; and the MBR has 16 bits, in order to contain a memory word.

The target level is stack-oriented, with arithmetic instructions that take their operands from the stack and push their results onto the stack. Figure 4-8 shows the target level's organization. There are two instruction formats, one for one-address instructions and one for 0 address instructions. The instruction set consists of 12 instructions, as shown in Fig. 4-9. There are one-address instructions to push data onto the stack and pop data off the stack. There is an unconditional jump, plus three kinds of conditional jumps. The conditional jumps pop the top word from the stack, test it to see if it is negative, zero, or positive (including zero), and jump if the specified condition is met. The procedure call instruction pushes the return address onto the stack and then jumps to the procedure located at the address specified in the instruction.

The instructions for addition, subtraction, multiplication, and division first pop their right operand and then their left operand from the stack. Next, they compute the result and finally, push it back onto the stack. If T is the top word of the stack and S is the word just below it, these instructions compute

S + T, S − T, S × T, and S/T

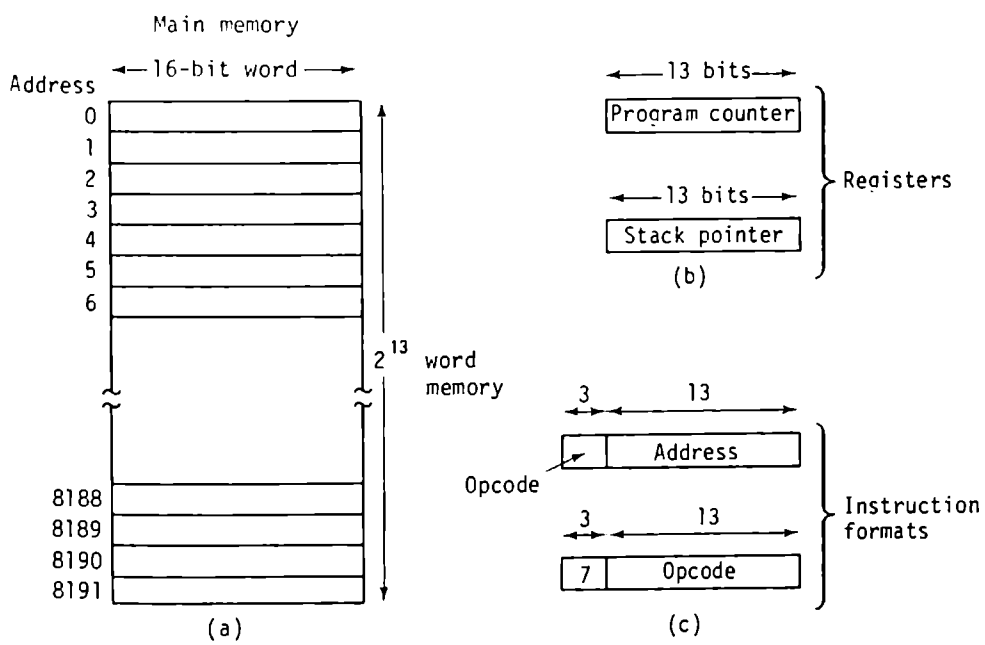


Fig. 4-8. The stack machine to be used in subsequent examples. (a) The 8K main memory. (b) The registers. (c) The instruction formats.

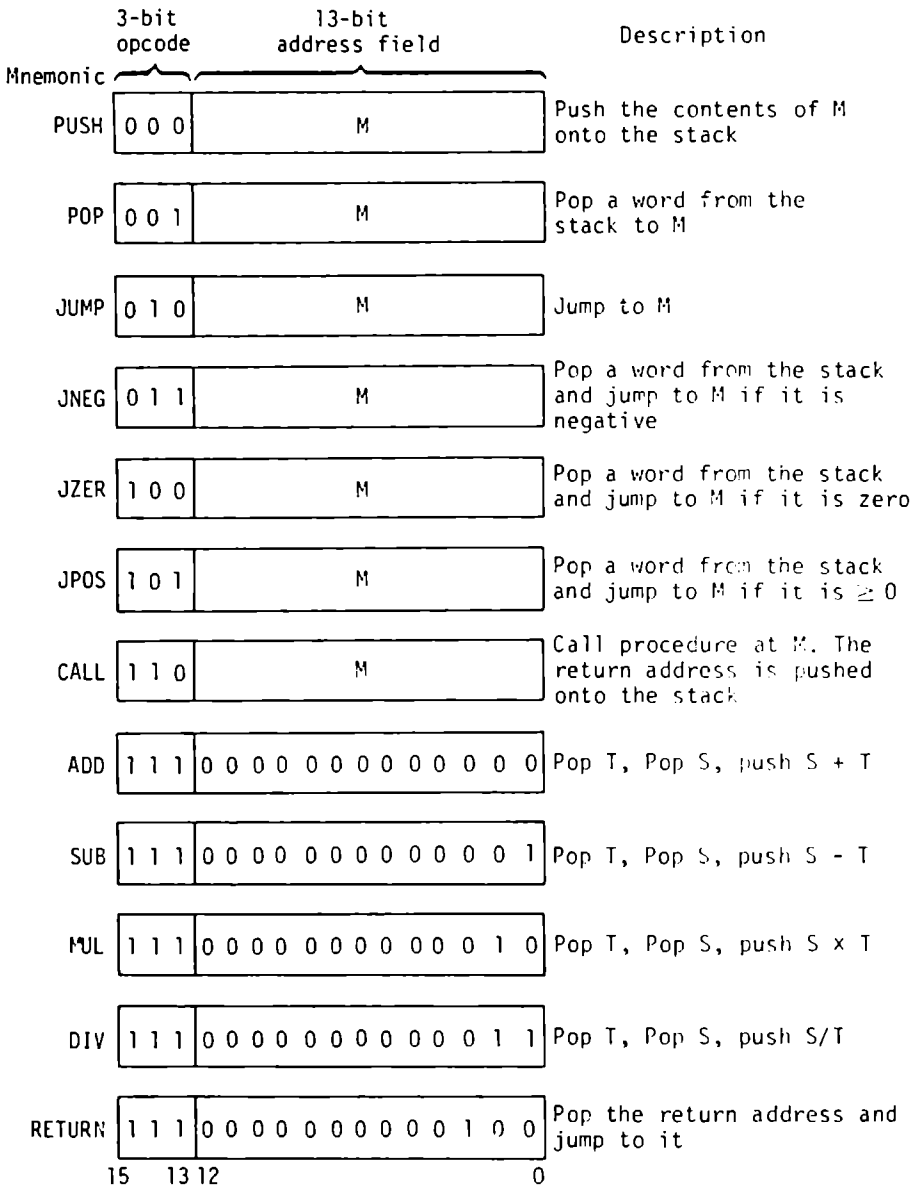

3-bit opcode
13-bit address field
Description
Mnemonic
PUSH 0 0 0 M Push the contents of M onto the stack
POP 0 0 1 M Pop a word from the stack to M
JUMP 0 1 0 M Jump to M
JNEG 0 1 1 M Pop a word from the stack and jump to M if it is negative
JZER 1 0 0 M Pop a word from the stack and jump to M if it is zero
JPOS 1 0 1 M Pop a word from the stack and jump to M if it is ≥ 0
CALL 1 1 0 M Call procedure at M. The return address is pushed onto the stack
ADD 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 Pop T, Pop S, push S + T
SUB 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 1 Pop T, Pop S, push S - T
MUL 1 1 1 0 0 0 0 0 0 0 0 0 0 0 1 0 Pop T, Pop S, push S x T
DIV 1 1 1 0 0 0 0 0 0 0 0 0 0 0 1 1 Pop T, Pop S, push S/T
RETURN 1 1 1 0 0 0 0 0 0 0 0 0 0 1 0 0 Pop the return address and jump to it
15 13 12 0

Fig. 4-9. The instruction set for the stack machine of Fig. 4-8. M is a memory address, T is the word on the top of the stack, and S is the word just below T on the stack.

respectively. Note that although the order of S and T is immaterial for addition and multiplication, it is important for subtraction and division. Each of these instructions pops two words from the stack and pushes one word onto the stack, making a net loss of one word. As words are pushed onto the stack, they are put into successively higher-numbered addresses (unlike the PDP-11). All arithmetic is performed in 2's complement.

The return instruction pops the return address off the stack and jumps there. It is used to return from procedures called with the call instruction.

The instructions shown in Fig. 4-9 use a simple expanding opcode scheme. There are potentially 8192 zero-address instructions using opcode 7. Five are actually used in this machine. Designing and implementing the other 8187 zero-address instructions will be left as an exercise for the reader.

4.4. A HYPOTHETICAL HOST LEVEL

In order to implement the 12-instruction, 16-bit-word target machine described above, we will use the microprogramming level organization shown in Fig. 4-10. This host level architecture will be used throughout this chapter as an example. It is composed of 15 registers, 39 data paths with gates, an adder for 16-bit binary numbers, and a 1-bit shifter.

4.4.1. The Host Level's Registers

The functions of the registers are as follows:

PC —Holds the target machine's program counter (13 bits)

SP —Holds the target machine's stack pointer (13 bits)

IR —Holds the target machine instruction being executed (16 bits)

MAR—Address register for the target machine's memory (13 bits)

MBR—Memory buffer register for target machine's memory (16 bits)

A —Temporary storage for the microprogram (16 bits)

B —Temporary storage for the microprogram (16 bits)

C —Temporary storage for the microprogram (16 bits)

D —Temporary storage for the microprogram (16 bits)

X —Counter used by the microprogram to execute loops 16 times (16 bits)

These registers are all read/write. They may serve both as the source and as the destination in register transfers. In addition, there are five read only registers that contain constants. These registers may only be used as sources and not destinations. The five constants are: +1, 0, −1, 15, and −32768. The register containing −32768 is labeled SIGN in Fig. 4-10 because −32768 in binary is

1000000000000000

that is, just the sign bit. This constant is needed to perform 32-bit shift operations.

All the registers are 16 bits wide except for those used to hold target machine addresses. These registers, PC, SP, and MAR, are only 13 bits wide because target machine addresses are only 13 bits.

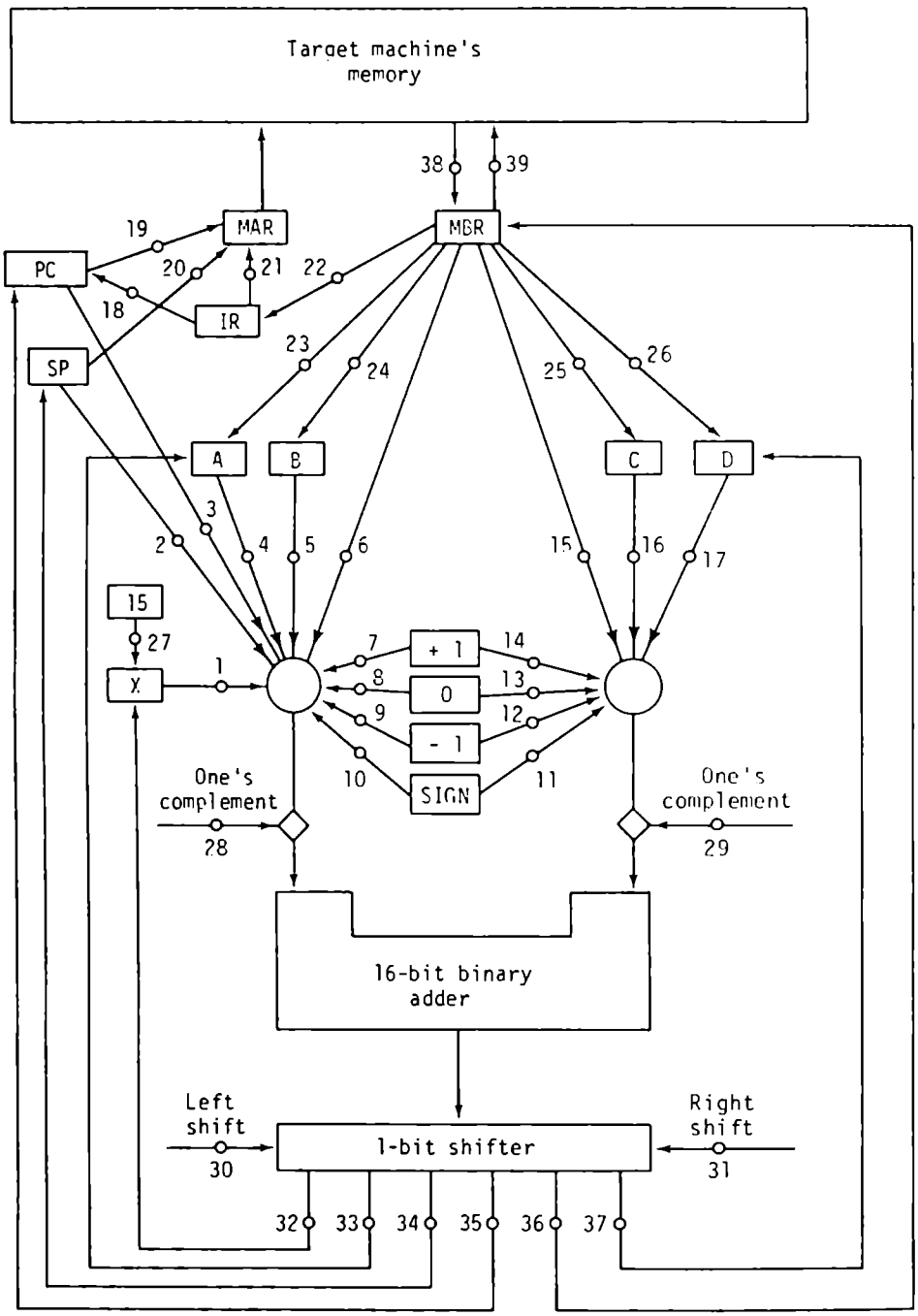


Fig. 4-10. The organization and data paths of the microprogramming level (host machine).

4.4.2. The Host Level's ALU

The CPU contains an adder that takes two 16-bit numbers as input and produces a single 16-bit number as output. The output is the 2's complement sum of the inputs. Overflow is not detected. The output of the adder is fed directly into a shift unit that can shift it left or right one bit. On left shifts, bit 0 is cleared to 0. On right shifts, bit 15 is cleared to 0—that is, no sign extension is performed. The operation of the adder and shifter is illustrated below in binary.

	Example 1	Example 2	Example 3
left input	0000000000100100	1111111111111111	0000000000000011
right input	0000000000000111	1111111111111111	1111111111111101
sum	0000000000101011	1111111111111110	0000000000000000
sum left shifted	0000000001010110	1111111111111100	0000000000000000
sum right shifted	0000000000010101	0111111111111111	0000000000000000

4.4.3. The Host Level's Gates and Data Paths

The host machine has 39 gates, numbered 1 to 39, as shown in Fig. 4-10. Some of these gates control register-to-register transfers (e.g., 18, 19, 20), while others control input to the adder (e.g., 1, 2, 3). Still others control output from the adder (e.g., 32, 33, 34). Gates 28, 29, 30, 31, 38, and 39 have special functions. The function of each gate is discussed below. The numbers refer to the gate numbers in the figure.

Adder Inputs

Gates 1 to 10 are for the left input; 11 to 17 are for the right input.

1. X to adder —used to subtract 1 from X for counting loop iterations.
2. SP to adder —used to add +1 or −1 to SP for stack operations.
3. PC to adder —used to route the PC to memory via MBR.
4. A to adder —used for arithmetic and transfers to D and MBR.
5. B to adder —used for arithmetic and transfers to A, D, and MBR.
6. MBR to adder—used for arithmetic.
7. +1 to adder —used to add 1 to a register.
8. 0 to adder —e.g., to move C to A, compute 0 + C, send result to A.
9. −1 to adder —used to subtract 1 from a register.
10. SIGN to adder—used for inserting carry bits on 32-bit shifts.
11. SIGN to adder—see 10. SIGN may be used as either left or right input.
12. −1 to adder —see 9. −1 may be used as either left or right input.

13. 0 to adder —see 8. Used to transfer A or B to D or MBR.
14. +1 to adder —see 7. +1 may be used as either left or right input.
15. MBR to adder—see 6. MBR is available as either input
16. C to adder —used for arithmetic and transfers to A, D, and MBR.
17. D to adder —used for arithmetic and transfers to A and MBR.

Register-to-Register Transfers

18. IR to PC —used in jump instructions to load new PC from the 13 bits or the IR. This data path is only 13 bits wide.
19. PC to MAR—used to fetch target machine instructions.
20. SP to MAR—used to fetch or replace top of the stack.
21. IR to MAR—used by PUSH and POP to fetch and store operands. This data path is only 13 bits wide.
22. MBR to IR —after being fetched, target machine instructions are moved to IR.
23. MBR to A —load the A register.
24. MBR to B —load the B register.
25. MBR to C —load the C register.
26. MBR to D —load the D register.
27. 15 to X —used to initialize X to 15 for looping.

Complementing and Shifting

28. When this gate is open, the 1's complement of the left adder input is used instead of the left input itself.
29. When this gate is open, the 1's complement of the right adder input is used instead of the right input itself.
30. When this gate is open, the adder output is shifted left one bit before being routed to the destination register.
31. When this gate is open, the adder output is shifted right one bit before being routed to the destination register.

Adder Output

32. output to X —used for counting from 15 to −1 for loops.
33. output to A —used for sending results to the A register.
34. output to SP —used for adding +1 or −1 to SP when pushing or popping.
35. output to PC —used for advancing PC to next instruction.
36. output to MBR —all results to be stored in memory must get to MBR via this data path.

37. output to D —used for sending results to D register.

Reading and Writing from the Target Machine's Memory

38. MBR to memory—used to load the MBR from the memory word specified by the MAR.
39. memory to MBR—used to store the MBR into the memory word specified by the MAR.

Although it is perfectly reasonable to gate the contents of a register into two or more other registers at the same time, it is not reasonable to load a register from two different sources simultaneously. For example, gates 23, 24, 25, and 26 could all be open simultaneously to copy MBR into A, B, C, and D during one cycle; but if gates 19, 20, and 21 were all open simultaneously, the results would be undefined. Similarly, if any of the output gates (32 to 37) are open, the adder must have exactly one left input and one right input, no more and no fewer. Furthermore, gates 30 and 31 may both be closed; but if one is open, the other must be closed.

4.5. GATE SEQUENCES

In order to see how the target level's programs are executed interpretively, we will consider the stack ADD instruction in detail. This instruction can be broken up into four steps:

1. Pop the first operand from the stack into one of the host level's registers.
2. Pop the second operand from the stack into another host level register.
3. Form the sum of the two operands.
4. Push the sum onto the stack.

All these operations can be performed by specifying the precise opening and closing sequences for the gates of Fig. 4-10.

The ADD instruction can be carried out in three cycles. Each cycle is initiated by a pulse from the clock. During each cycle, certain gates will be opened and the others will remain closed. If gates were opened during the entire cycle, certain ambiguities could result. For example, if gates 4, 13, 36, and 39 were opened during the same cycle, would the A register be copied into the MBR before, during, or after the MBR was stored into memory? In other words, would A be stored or would the original MBR value be stored?

4.5.1. Subcycles

In order to resolve this kind of difficulty, we will divide each host level cycle into three subcycles. During the first subcycle, gates 1 to 29 may be opened. During the second subcycle, gates 30 to 37 may be opened. During the third and last subcycle, gate 38 or 39 (not both) may be opened.

Conceptually, one may imagine that the adder has two internal registers, left input and right input, which are loaded during the first subcycle, depending on the status of gates 1 to 17, 28, and 29. The first subcycle is for register-to-register transfers. During the second subcycle, the adder and shifter operate, and the output is gated back into the registers selected by gates 32 to 37. Finally, during the third subcycle, the MBR is loaded or stored if gate 38 or 39 is open, respectively. Of course, both may be closed, in which case no memory transfer occurs during that cycle.

The decision to break the cycle into subcycles for register-to-register transfers, adder-shifter operations, and memory transfers resolves the ambiguity posed above (having gates 4, 13, 36, and 39 open during one cycle). During subcycle 1, the adder inputs, A and 0, are prepared. During subcycle 2, the sum A + 0 is computed and stored in the MBR. During subcycle 3, the MBR (now containing A) is stored into the memory word specified by the MAR.

In this particular design, a memory transfer can be completed during one subcycle. This is a simplification of the real world, in order to make the host level easier to understand. In actual practice, the main memory cycle time may be one or more processor cycles. The complications introduced by main memories with long cycle times will be described in Section 4.9.3.

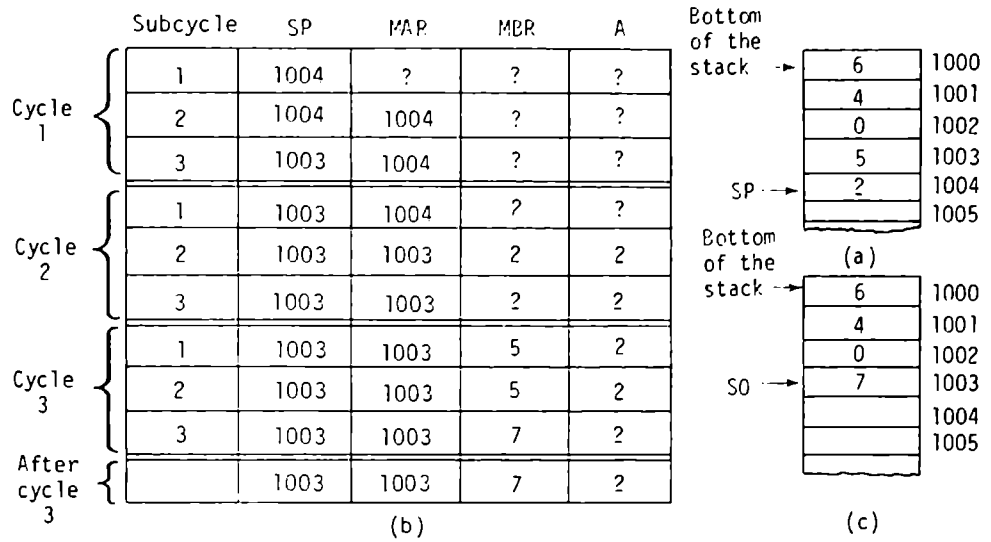
4.5.2. Gate Sequences for the ADD Instruction

Let us now return to the interpretation of the target machine's ADD instruction. Assume that the ADD instruction is already present in the IR and that previous cycles have determined that it is an ADD instruction. The gates opened during the ADD instruction's three cycles are as follows:

cycle 1: gates 2, 12, 20, 34, 38
cycle 2: gates 20, 23, 38
cycle 3: gates 4, 15, 36, 39

In cycle 1 the first operand is fetched into MBR by gating SP into MAR (gate 20) and starting a memory read (gate 38). The SP register contains the memory address of the top of the target machine's stack; therefore gating SP into MAR and then starting a read cycle fetches the top word of the stack into the MBR. Concurrent with fetching the top of the stack, SP is decremented by 1 by adding it to −1 and gating the result back into SP (gates 2, 12, 34). Because the cycle is divided into subcycles, MAR is loaded from SP during subcycle 1, and the new value of SP is not stored into SP until subcycle 2. Figure 4-11 illustrates the addition of 2 and 5 with the 2 on the top of the stack in location 1004 and 5 just below it in location 1003.

During cycle 2, the first operand is saved in A, and the other operand is loaded into MBR. Note that SP is not decremented again during the second cycle because the result will be stored back in place of the second operand. If SP were decremented during cycle 2, it would have to be incremented again later before storing the result back. Eliminating the redundant decrement-increment operations saves one cycle.



	Subcycle	SP	MAR	MBR	A
Cycle 1	1	1004	?	?	?
	2	1004	1004	?	?
	3	1003	1004	?	?
Cycle 2	1	1003	1004	?	?
	2	1003	1003	2	2
	3	1003	1003	2	2
Cycle 3	1	1003	1003	5	2
	2	1003	1003	5	2
	3	1003	1003	7	2
After cycle 3		1003	1003	7	2

Fig. 4-11. (a) Stack configuration before the ADD instruction. (b) The values of SP, MAR, MBR, and A directly before the start of each subcycle. (c) The stack configuration after the ADD instruction.

During cycle 3, the addition of the operands is performed and the sum routed back into MBR (gates 4, 15, 36). Finally, the sum is stored back into memory, overwriting the second operand (gate 39). The SP register is not changed during cycle 3, with the result that the target machine's ADD instruction has a net effect of reducing the number of words on the stack by one.

Figure 4-12 shows the sequence of gates that are opened during the interpretation of the ADD instruction. The gates not shown are closed throughout.

One technique for building a computer is to build specific circuits (timing rings) that produce gate sequences like those of Fig. 4-12. When a pulse is applied to a timing ring, the sequencing starts and for the next few cycles the status of all the gates is controlled by the pulses from the timing ring. The predecessors of most present-day computers used this principle. Some still use it even now.

Despite its former widespread use, this technique has some serious disadvantages. If the machine has many gates and many instructions, the circuitry needed to produce the proper sequences for all the gates during all the instructions will be very complicated. The result is to make the machine expensive, unreliable, and difficult to repair.

On the other hand, if the designers want to squeeze every last nanosecond out of the machine—that is, if the rate at which it performs floating-point multiplications has precedence over all other criteria (the CDC Cyber 76 comes close to meeting this condition)—then this is still the way to do it.

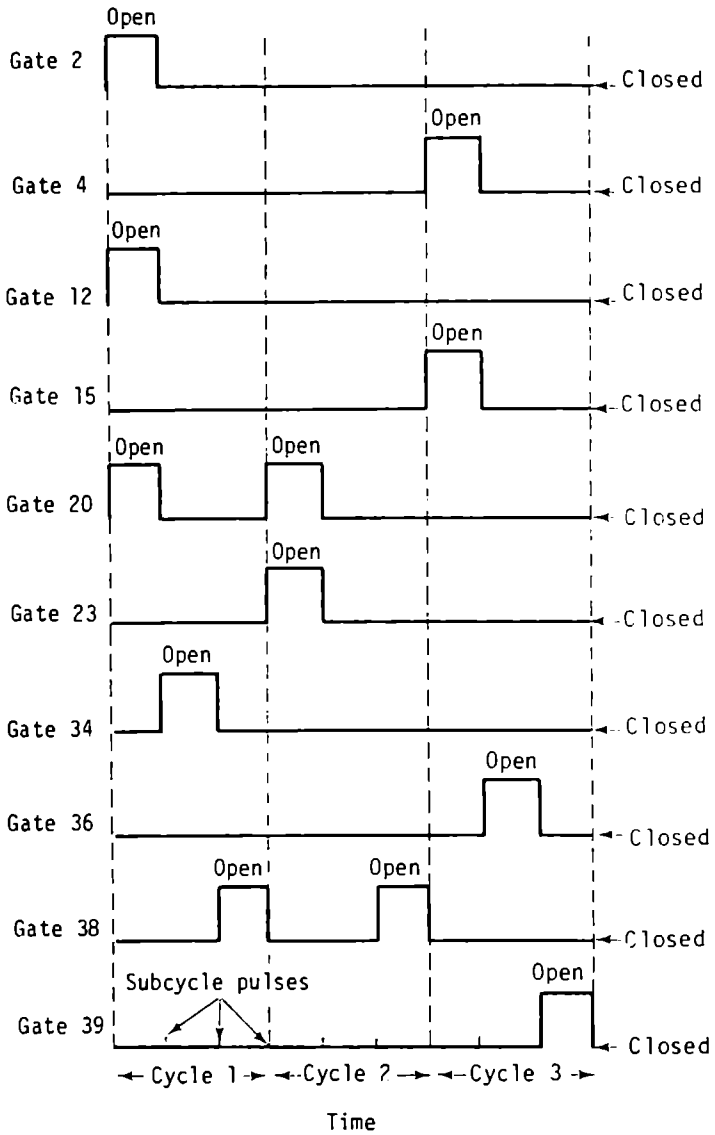


Fig. 4-12. The ADD instruction is accomplished by opening and closing these ten gates during the subcycles indicated.

4.6. MICROPROGRAMMED GATE CONTROL

Microprogramming is an alternative approach to controlling gate sequences. The basic idea, first suggested by Wilkes in 1951 (reprinted in Bell and Newell, Chapter 28), is quite straightforward. In order to cause the target machine to execute its program, it is necessary to open and close gates in a certain sequence. The

description of which gates to open when constitutes a program for a machine whose instruction set includes an "open gate" instruction.

Up until now we have not specified the exact format of the instructions executed by our host level, nor have we specified where they are stored nor how they are executed. These subjects will now be examined.

4.6.1. Microinstructions

The host level (Fig. 4-10) has an instruction set consisting of only two microinstructions. These instructions, GATE and TEST, are shown in Fig. 4-13. Because there are only two instructions, a one-bit opcode is sufficient. Bit 0 specifies the instruction: GATE = 1 and TEST = 0. Note that the opcode is the rightmost bit rather than the leftmost bit. Each of the remaining 39 bits in the GATE microinstruction controls one of the CPU's 39 gates. A 1 means gate on, and a 0 means gate off. In order to accomodate all the bits, the microinstructions are 40 bits long.

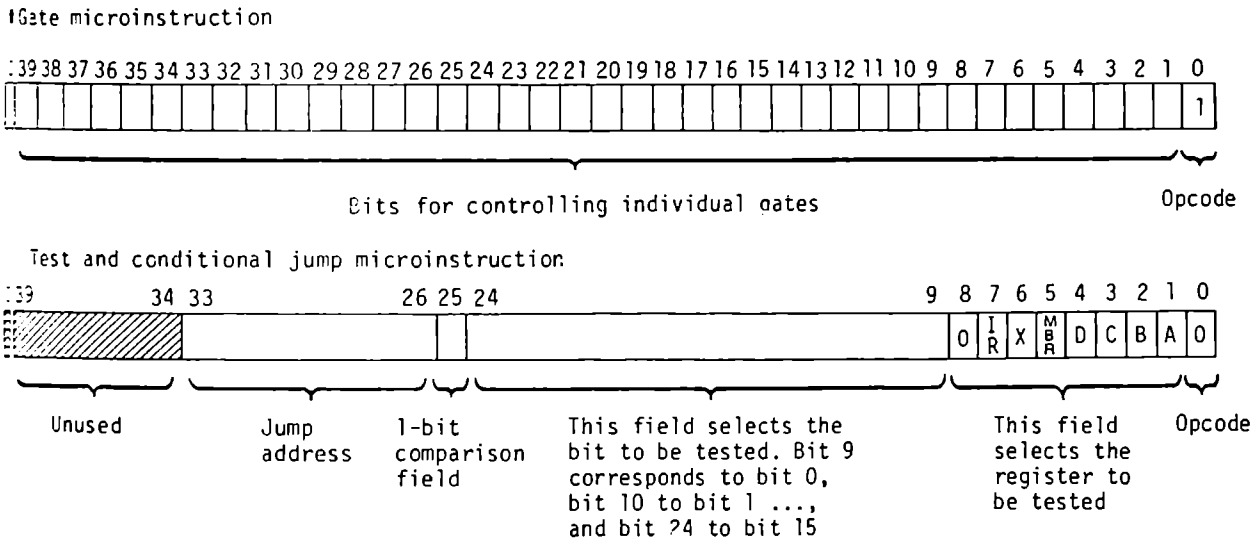

Gate microinstruction
39 38 37 36 35 34 33 32 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1
Bits for controlling individual gates
Opcode
Test and conditional jump microinstruction
39 34 33 26 25 24 9 8 7 6 5 4 3 2 1 0
0 IR X MBR D C B A 0
Unused
Jump address
1-bit comparison field
This field selects the bit to be tested. Bit 9 corresponds to bit 0, bit 10 to bit 1 ..., and bit 24 to bit 15
This field selects the register to be tested
Opcode

Fig. 4-13. The instruction set for the microprogramming level of Fig. 4-10 has only two instructions.

The TEST microinstruction is used to conditionally alter the sequence of microinstructions executed. Each TEST microinstruction specifies one bit to be tested, out of 128 possibilities. If the selected bit is the same as bit 25 of the microinstruction, the sequence of microinstructions to be executed is changed, and the address of the next microinstruction is specified by bits 26 to 33 of the TEST microinstruction itself. If the selected bit is different from bit 25, execution just continues with the following microinstruction.

The bit to be tested is specified by two fields within the microinstruction. Bits

1 to 8 specify which host machine register is to be tested according to the following rules:

BIT 1—A register

BIT 2—B register

BIT 3—C register

BIT 4—D register

BIT 5—MBR register

BIT 6—IR register

BIT 7—X register

BIT 8—register containing the constant 0 (all of whose bits are zero)

Bits 9 to 24 of the microinstruction tell which bit within the selected register to test, with microinstruction bit 9 meaning register bit 0, microinstruction bit 10 meaning register bit 1, and so on. Exactly one bit in the group 1 to 8 must be set to 1, and exactly one bit in the group 9 to 24 must be set to 1.

4.6.2. Execution of Microprograms

The microprogram is kept in a **micromemory** (or **control store**) that is logically distinct from the memory used by the target machine's programs and data. On some computers there is only one physical memory, which is partitioned into two parts—one for the target machine's program and data and one for the microprogram. Because the GATE microinstruction requires 40 bits, the micromemory consists of 40-bit words. The maximum size of the micromemory is determined by the number of address bits available in the TEST instruction. The format of Fig. 4-13 could provide for up to 14-bit addresses, which puts an upper limit of 2^{14} words in the micromemory. (We could call them microwords, but 40 bits is hardly "micro.") For the simple interpreter needed in this example, 256 words of micromemory will be more than adequate and will even leave room for future expansion of the interpreter. Micromemory addresses will be eight bits, and bits 34 to 39 of the TEST microinstruction will not be used.

In order to control the execution of the microprogram, a **microprogram counter** is needed, analogous to the conventional machine level's program counter. To fetch microinstructions from the micromemory, an MAR and a MBR are needed. Because the machine is so simple, the microprogram counter and the MAR are the same. Figure 4-14 shows the complete architecture of the microprogramming level.

The operation of the CPU is as follows. When a clock pulse arrives, a new cycle begins. The micromemory word specified by the microprogram counter (which also serves as the micromemory's MAR) is fetched and loaded into the microinstruction register (which also serves as the micromemory's MBR). If it is a GATE microinstruction, those gates specified by bits 1 to 39 are opened during the ap-

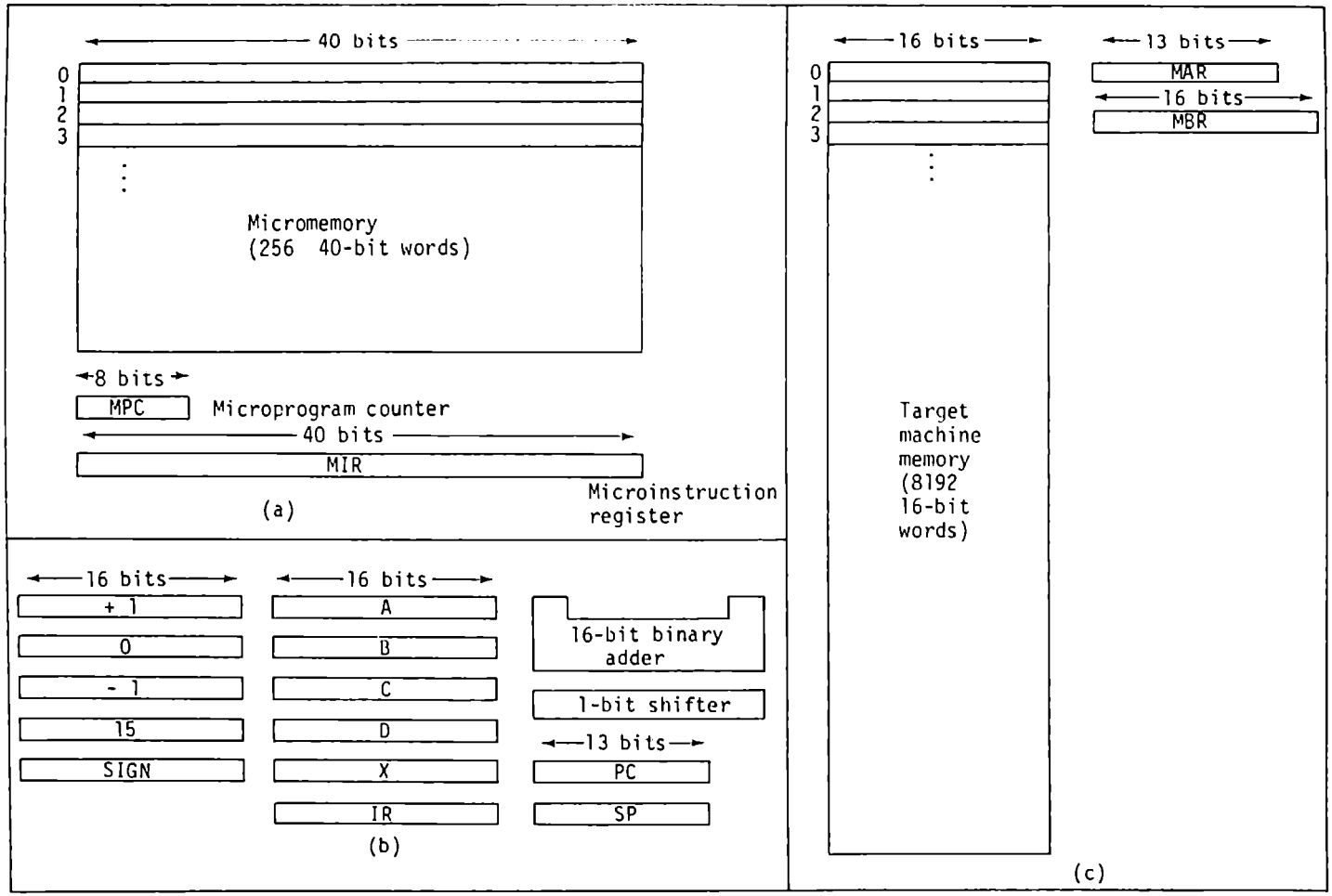


Fig. 4-14. The complete host machine hardware. (a) The micromemory and its controlling registers. (b) The working registers and functional units (see Fig. 4-10 for the data paths). (c) The main memory and its controlling registers.

propriate subcycle. If it is a TEST microinstruction, the selected bit is located and tested. If it matches bit 25 of the microinstruction register, bits 26 to 39 of the microinstruction are gated into the microprogram counter; otherwise execution continues. Each microinstruction requires one cycle to be executed.

It should be clearly realized that the microprogrammed computer described here has 2 memories, 2 instruction sets, and 2 programs: the conventional machine level's and the microprogramming level's. There is, however, only one real hardware processor, and its architecture forms the microprogramming level.

The conventional machine level is the one described in the manufacturer's manuals. These manuals would advertise the computer as having a memory of 8192 16-bit words, and an instruction set consisting of four stack arithmetic operations—add, subtract, multiply, and divide—and eight other instructions—push, pop, jump, three conditional jumps, call, and return—as shown in Fig. 4-9. The manual might mention that the machine was microprogrammed, but it would not discuss the microprogramming level, the micromemory, or the microinstructions.

A person who wants to program at the conventional machine level, after all, does not care whether the manufacturer has chosen to implement it by building actual circuits to control the gates or whether the gates are under the control of a microprogram running at a lower level. In fact, he need not even be concerned with the gates at all. The microprogrammer (the person who wrote the microprogram), on the other hand, has to know all about both the conventional machine level and the microprogramming level, including all the gates and data paths.

4.6.3. A Two-Level Machine

This example shows a two-level machine organization. A programmer who knew about the microprogramming level, and who was allowed to and capable of changing its microprogram, could decide to write his program in two different ways. The first way would be to write it as a series of target machine instructions, using the instructions in Fig. 4-9. This is the normal approach that almost everyone uses.

A second alternative would be first to write a new microprogram—in other words, to define new instructions for the conventional machine level—and then write the program as a series of these newly defined target level instructions. The choice of which way to do it is a matter of cost, convenience, and policy (the computer center or manufacturer might not like applications programmers to write new microprograms) more than anything else. Conceptually, both approaches are equally valid.

An analogy can be made between the target level's instructions and procedures in programming languages. A procedure consists of a number of statements in the programming language, designed to perform some particular function. The programmer could first define a collection of procedures to do tasks that are important to his problem and then write his program as a series of calls to these procedures. This technique simplifies the programming by dividing it into two parts: writing the procedures and combining them to form the program.

The programmer who decides to write a new microprogram first, in order to create a new target machine instruction set, is doing something similar. He could have written the entire program directly as a microprogram, but it is much more convenient first to create operations like add and multiply and then to construct the program from them rather than from gate sequences.

4.7. A LANGUAGE FOR MICROPROGRAMMING

Although it is possible to write microprograms as sequences of 40-bit binary numbers, doing so is both tedious and error prone. A microprogramming language better suited to human beings is needed. In this section a simple MicroProgramming Language, MPL, based on PL/I, will be developed for microprogramming our example computer.

One difference between microprogramming and conventional machine level programming is the degree of parallelism present. A single microinstruction may cause several register transfers to occur simultaneously, and this fact must be reflected in the microprogramming language. By definition, each line of MPL will represent a single microinstruction, even though it contains several statements. Statements on the same line are executed in parallel.

4.7.1 Notation for GATE Microinstructions

Register-to-register transfers will be indicated by assignment statements. For example,

```
A = MBR;
```

means gate 23 is to be opened during that microinstruction. Statements are terminated by semicolons. The order of statements on a line is unimportant. The two lines

```
A = MBR; MBR = IR; X = 15;
MAR = IR; X = 15; A = MBR;
```

will each produce the same microinstruction—namely, a GATE instruction with bits 21, 23, and 27 set. Both statements differ from

```
A = MBR;
MAR = IR;
X = 15;
```

which compiles into three successive microinstructions.

To indicate the use of the adder, the result register should be written as the sum of the operands. Some examples are as follows:

```
MBR = A + C;              /* GATES 4, 16, 36 */
SP = SP + (-1);           /* GATES 2, 12, 34 */
A = MBR; PC = PC + 1;     /* GATES 3, 14, 23, 35 */
D = 0 + C;                /* GATES 8, 16, 37 */
```

Comments are enclosed between /* and */. Note that the last line serves to copy C into D via the adder, even though there is no direct data path.

The 1's complementors, controlled by gates 28 and 29, are indicated by writing the operand to be complemented as an argument of the COM function. For example,

```
MBR = COM(MBR) + 1;          /* GATES 6, 14, 28, 36 */
A = 0 + COM(SIGN);           /* GATES 8, 11, 29, 33 */
```

To indicate left and right shifting, the operands are written as arguments of the functions LEFT_SHIFT and RIGHT_SHIFT, respectively. For example,

```
MBR = LEFT_SHIFT(1 + 1);          /* GATES 7, 14, 30, 36 */
A   = RIGHT_SHIFT(A + 0);         /* GATES 4, 13, 31, 33 */
```

4.7.2 Notation for TEST Microinstructions

TEST microinstructions are indicated by writing

```
IF BIT(k,reg) = b THEN GO TO label;
```

where *k* is the bit number to be tested, 0 to 15; reg is the register to be tested, A, B, C, D, MBR, IR, X, or 0; b is 0 or 1; and label is the microinstruction to be jumped to if the test succeeds.

A line may begin with a symbolic label followed by a colon. Note that testing any bit of the constant register 0 to see if it contains a 0 always succeeds, yielding an unconditional jump, which can be abbreviated

```
GO TO label;
```

Some examples of TEST microinstructions are

```
IF BIT(13,IR) = 1 THEN GO TO POP;  /* TESTS BIT 13 OF IR */
IF BIT(15,X) = 0 THEN GO TO MULLOOP;  /* TESTS BIT 15 OF X */
GO TO MAINLOOP;        /* TESTS THE 0 REGISTER */
```

Because the use of GO TO statements is generally considered poor programming practice, it is important to understand why they have been included in this microprogramming language. When writing applications programs in problem-oriented languages, such as ALGOL 68, FORTRAN, PL/I, or PASCAL, the programmer usually does not care what instructions the compiler generates. His main concern is writing correct, well-structured programs.

In contrast, our goal is to see how particular combinations of microinstructions can be used to interpret particular target machine instructions. The details of the "generated code" are very much of concern to us, so we need a language in which the programmer can "force" the translator to produce any valid microinstruction he wants. MPL is, in fact, an assembly language because "you get what you write"; that is, there is a one-to-one correspondence between source statements and generated microinstructions.

4.8. THE INTERPRETER FOR THE TARGET MACHINE

Figure 4-15 is the complete interpreter for the target machine. You should study it carefully before proceeding. The label MAINLOOP is at the start of the fetch instruction, examine instruction, and execute instruction loop common to all interpreters. Lines 0 and 1 fetch the target machine instruction, move it to IR for analysis, and advance PC by 1. Lines 2, 3, 4, 11, 17, 18, 28, 32, 33, and 40 constitute a decoding tree that tests bits 0, 1, 2, 13, 14, and 15 of IR to determine the opcode type and jump to the appropriate part of the microprogram for subsequent processing. The decoding logic is shown in Fig. 4-16. Three tests are needed to determine that an instruction is a PUSH, whereas six tests are needed for a DIV. Note that ADD and SUB share a common section of code due to their similarity.

Although this decoding method is very general, being able to analyze any possible opcode system, it is time consuming, requiring one test per opcode bit. For a target machine such as the 370, with an eight-bit opcode, eight cycles would be needed. A faster decoding method can be used on host machines in which the microprogram counter itself can be gated into and out of. By adding the opcode to the microprogram counter, a many-way jump can be achieved. Each of the addresses jumped to can contain a jump to the interpreting code for the corresponding instruction.

The PUSH and POP instructions are straightforward, each requiring three microinstructions, the last of which is a jump back to the main loop. The conditional jumps that pop a word off the stack and test its sign bit, JNEG and JPOS, also require three microinstructions. Both instructions remove the tested word from the stack. If the target machine instructions had not required popping a word off the stack, the interpreter would not have bits 2, 12, and 34 set in the microinstructions at words 14 and 25.

The interpretation of JZER is more complicated than that of JNEG or JPOS because it requires testing all 16 bits of the top of the stack, not just the sign bit. One way of testing all 16 bits would be to have 16 consecutive TEST microinstructions, each one testing a different bit. The result would be a faster method than the one used here, but it would take up more space in the micromemory. The method of Fig. 4-15 is to use the X register as a counter and execute a loop 16 times, each iteration testing the sign bit of MBR and then left shifting MBR so that the next bit will be tested the next time through the loop. The counter, X, starts at 15. The looping stops when the sign bit of X becomes a 1—that is, when X has decreased to -1. Sixteen iterations are required. The reason is that testing for -1 is easier than testing for 0 on this host machine. The microprogramming level of most commercially available computers has special hardware to make testing for 0 equally easy.

The interpretation of ADD and SUB is noteworthy only because the same code is used for both. The difference $P - Q$ is computed as $P + (-Q)$. Line 35 tests to see if the instruction is an ADD and, if not, negates the second operand (line 36) before forming the sum.

```
ADDRESS         STATEMENT

    0    MAINLOOP:  MAR= PC; MBR= MEMORY (MAR); /* FETCH NEXT TARGET INSTR */
    1               IR= MBR; PC= PC + 1; /* MOVE INSTR TO IR AND ADVANCE PC */
    2               IF BIT(15,IR)= 1  THEN GO TO OP4567; /* DETERMINE TYPE */
    3               IF BIT(14,IR)= 1  THEN GO TO OP23;
    4               IF BIT(13,IR)= 1  THEN GO TO POP;

    5    PUSH:      MAR=IR; SP= SP + 1; MBR= MEMORY (MAR);
    6               MAR =SP; MEMORY (MAR) = MBR;
    7               GO TO MAINLOOP;

    8    POP:       MAR = SP;  SP = SP + (-1); MBR = MEMORY(MAR);
    9               MAR=IR; MEMORY(MAR) = MBR;  /* MAR=IR IS 13 BITS WIDE */
   10               GO TO MAINLOOP;

   11    OP23:      IF BIT(13,IR) = 1  THEN GO TO JNEG;

   12    JUMP:      PC = IR;      /* THIS DATA PATH IS 13 BITS WIDE */
   13               GO TO MAINLOOP;

   14    JNEG:      MAR = SP;  SP = SP + (-1);  MBR = MEMORY(MAR);
   15               IF BIT(15,MBR) = 1  THEN GO TO JUMP;
   16               GO TO MAINLOOP;

   17    OP4567:    IF BIT(14,IR) = 1  THEN GO TO OP67;
   18               IF BIT(13,IR) = 1  THEN GO TO JPOS;

   19    JZER:      X = 15;  MAR = SP;  SP = SP + (-1);   MBR = MEMORY(MAR);
   20    JLOOP:     IF BIT(15,MBR) = 1  THEN GO TO MAINLOOP;
   21               MBR = LEFT_SHIFT(MBR + 0);
   22               X = X + (-1);
   23               IF BIT(15,X) = 0  THEN GO TO JLOOP;
   24               GO TO JUMP;

   25    JPOS:      MAR = SP;  SP = SP + (-1);  MBR = MEMORY(MAR);
   26               IF BIT(15,MBR) = 0  THEN GO TO JUMP;
   27               GO TO MAINLOOP;

   28    OP67:      IF BIT(13,IR) = 1  THEN GO TO OP7;

   29    CALL:      SP = SP + 1;
   30               MAR = SP;  MBR = PC + 0;  MEMORY(MAR) = MBR;
   31               GO TO JUMP;

   32    OP7:       IF BIT(2,IR) = 1  THEN GO TO RETURN;
   33               IF BIT(1,IR) = 1  THEN GO TO MULDIV;

   34    ADDSUB:    MAR = SP;  SP = SP + (-1);  MBR = MEMORY(MAR);
   35               IF BIT(0,IR) = 0  THEN GO TO SUM;
   36               MBR = COM(MBR) + 1;    /* FOR SUBTRACTION ONLY */
   37    SUM:       MAR = SP;  A = MBR;  MBR = MEMORY(MAR);
   38               MBR = A + MBR;  MEMORY(MAR) = MBR;
   39               GO TO MAINLOOP;
```

Fig. 4-15. The microprogram.

```
40   MULDIV:    IF BIT(0,IR) = 1  THEN GO TO DIV;

41   MUL:       MAR = SP;  SP = SP + (-1);  MBR = MEMORY(MAR);
42              C = MBR;  MAR = SP;  MBR = MEMORY(MAR);
43              X = 15;  A = 0 + 0;  D = 0 + 0;
44              IF BIT(15,MBR) = 0  THEN GO TO MULLOOP;
45              MBR = COM(MBR) + 1;
46              D = MBR;  MBR = 1 + COM(C);
47              C = MBR;  MBR = 0 + D;
48   MULLOOP:   IF BIT(0,MBR) = 0  THEN GO TO NOADD;
49              A = A + C;
50   NOADD:     MBR = RIGHT_SHIFT(MBR + 0);
51              D = RIGHT_SHIFT(0 + D);
52              IF BIT(0,A) = 0  THEN GO TO NOCARRY;
53              D = SIGN + D;
54   NOCARRY:   A = RIGHT_SHIFT(A + 0);
55              IF BIT(15,C) = 0  THEN GO TO MULEND;
56              IF BIT(14,A) = 0  THEN GO TO MULEND;
57              A = A + SIGN;
58   MULEND:    X = X + (-1);
59              IF BIT(15,X) = 0  THEN GO TO MULLOOP;
60              MBR = 0 + D;  MEMORY(MAR) = MBR;
61              GO TO MAINLOOP;
62   DIV:       MAR = SP;  SP = SP + (-1);  MBR = MEMORY(MAR);
63              C = MBR;  B = MBR;  MAR = SP;  A = 0 + 0;  MBR = MEMORY(MAR);
64              D = MBR;  MBR = COM (MBR) + 1;  X = 15;
65              IF BIT(15,MBR) = 1  THEN GO TO DVDPOS;
66              D = MBR;  MBR = COM(B) + 1;
67              B = MBR;
68   DVDPOS:    MBR = 1 + COM(C);
69              IF BIT(15,C) = 0  THEN GO TO DIVLOOP;
70              C = MBR;  MBR = COM(MBR) + 1;
71   DIVLOOP:   A = LEFT_SHIFT (A + 0);
72              IF BIT(15,D) = 0  THEN GO TO NOCARRY2;
73              A = A + 1;
74   NOCARRY2:  D = LEFT_SHIFT(0 + D);
75              A = A + MBR;
76              IF BIT(15,A) = 1  THEN GO TO DIVNEG;
77              D = D + 1;
78   DIVNEG:    IF BIT(15,A) = 0  THEN GO TO DIVPOS;
79              A = A + C;
80   DIVPOS:    X = X + (-1)
81              IF BIT(15,X) = 0  THEN GO TO DIVLOOP;
82              IF BIT(15,B) = 0  THEN GO TO DIVEND;
83              D = 1 + COM(D);
84   DIVEND:    MBR = 0 + D;  MEMORY(MAR) = MBR;
85              GO TO MAINLOOP;

86   RETURN:    MAR = SP;  SP = SP + (-1);  MBR = MEMORY(MAR);
87              IR = MBR;
88              GO TO JUMP;
```

Fig. 4-15. (cont.)

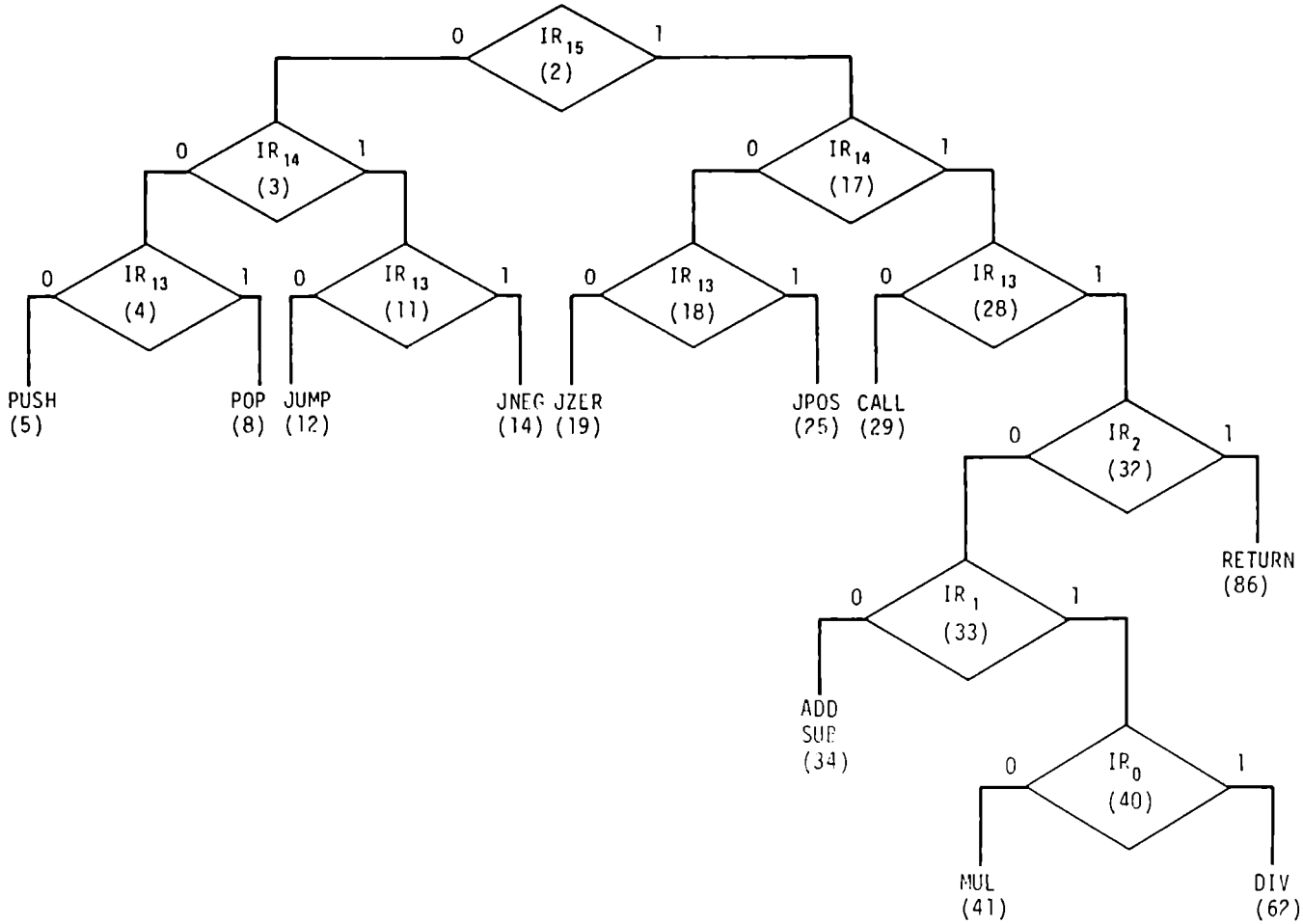


Fig. 4-16. The decoding tree used to determine which target machine instruction has been fetched. Each decision box shows the IR bit tested and the address of the microinstruction making the test. The labels and addresses of the routines that interpret specific target machine instructions are also shown.

4.8.1. Interpretation of the Multiplication Instruction

The algorithms for multiplication and division are the most complicated ones in the microprogram. The multiplication algorithm, which begins at the label MUL, is the binary equivalent of the algorithm everyone (hopefully) learns in elementary school. Figure 4-17 shows the multiplication of 69 × 25 in binary, using the familiar method of summing the shifted partial products. Each of the 16 partial products is formed by multiplying one digit (bit) of the multiplier by the multiplicand. For binary numbers, the multiplier digits are all 0 or 1; consequently, the partial products are either equal to the multiplicand or 0.

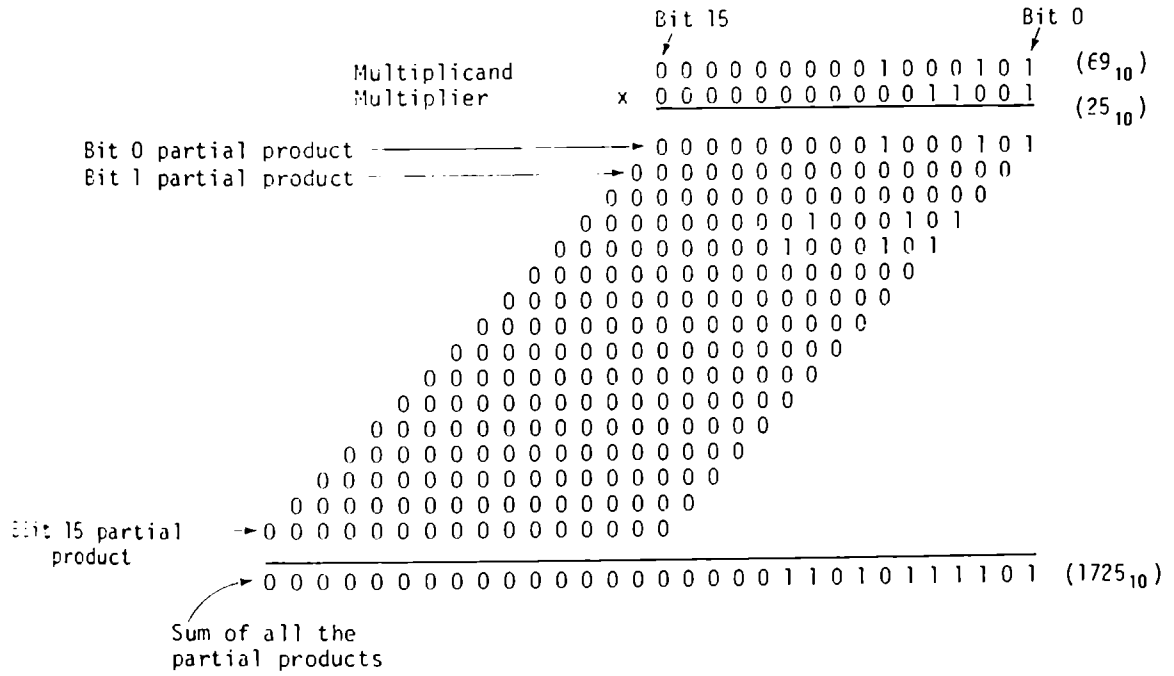


Fig. 4-17. Multiplication of 69 by 25 in binary.

One minor difference between the microprogrammed multiplication algorithm and that of Fig. 4-17 is that in the latter all 16 partial products are first computed and then their sum is formed. The computer algorithm adds up the partial products as it proceeds. Registers A and D form a 32-bit accumulator in which the sum of the partial products is kept. The 16 most significant bits are in A; the 16 least significant bits are in D. The multiplicand is kept in register C, and the multiplier is kept in MBR. Lines 41 to 47 initialize the registers. The multiplication loop, lines 48 to 59, is executed 16 times, once for each multiplier bit. X serves as a counter.

The loop begins by testing one bit of the multiplier (line 48). If the bit tested is a 0, there is no need actually to add 0 to the accumulator; therefore line 49 is skipped. If the bit tested is a 1, the multiplicand is added to the accumulator (line 49). Line 50 shifts the multiplier right one bit, in order to test successive bits on each iteration.

Rather than add the multiplicand to successively higher-order bit positions of the accumulator, it is more convenient always to add it to the same position and shift the contents of the 32-bit accumulator one bit to the right each time through the loop. This step is accomplished by lines 51 to 57, including the sign extension needed for negative multiplicands. Finally, X is decremented and tested, and the loop repeated.

Figure 4-18 shows the 32-bit accumulator formed by A and D, the multiplier in MBR, and the counter in X just before line 48 is executed each time through the loop. As an example, 69 is multiplied by 25, just as in Fig. 4-17. The multiplicand in C retains the value 69 throughout the calculation.

After the multiplication is completed, the low-order 16 bits are pushed onto the stack. If the product exceeds 16 bits, an undetected overflow condition occurs.

A	D	MBR	X
0000 0000 0000 0000	0000 0000 0000 0000	0000 0000 0001 1001	0000 0000 0000 1111
0000 0000 0010 0010	1000 0000 0000 0000	0000 0000 0000 1100	0000 0000 0000 1110
0000 0000 0001 0001	0100 0000 0000 0000	0000 0000 0000 0110	0000 0000 0000 1101
0000 0000 0000 1000	1010 0000 0000 0000	0000 0000 0000 0011	0000 0000 0000 1100
0000 0000 0010 0110	1101 0000 0000 0000	0000 0000 0000 0001	0000 0000 0000 1011
0000 0000 0011 0101	1110 1000 0000 0000	0000 0000 0000 0000	0000 0000 0000 1010
0000 0000 0001 1010	1111 0100 0000 0000	0000 0000 0000 0000	0000 0000 0000 1001
0000 0000 0000 1101	0111 1010 0000 0000	0000 0000 0000 0000	0000 0000 0000 1000
0000 0000 0000 0110	1011 1101 0000 0000	0000 0000 0000 0000	0000 0000 0000 0111
0000 0000 0000 0011	0101 1110 1000 0000	0000 0000 0000 0000	0000 0000 0000 0110
0000 0000 0000 0001	1010 1111 0100 0000	0000 0000 0000 0000	0000 0000 0000 0101
0000 0000 0000 0000	1101 0111 1010 0000	0000 0000 0000 0000	0000 0000 0000 0100
0000 0000 0000 0000	0110 1011 1101 0000	0000 0000 0000 0000	0000 0000 0000 0011
0000 0000 0000 0000	0011 0101 1110 1000	0000 0000 0000 0000	0000 0000 0000 0010
0000 0000 0000 0000	0001 1010 1111 0100	0000 0000 0000 0000	0000 0000 0000 0001
0000 0000 0000 0000	0000 1101 0111 1010	0000 0000 0000 0000	0000 0000 0000 0000

Fig. 4-18. The values of A, D, MBR, and X at the line labelled MULLOOP in Fig. 4-15 each time through the loop during the multiplication of 69 by 25.

4.8.2. Interpretation of the Division Instruction

Just as multiplication is carried out by adding and shifting, division is carried out by subtracting and shifting. In the fraction A/B, A is the dividend and B is the divisor. One problem present in division is determining how many times to subtract the divisor from the dividend. To clarify this point, you should divide 1749598 by 1944 before reading further. The first step is estimating "how many 1944s are there

in 17495?" Making this estimate accurately is not easy for most people. Nor is it easy for most computers.

After making an estimate of the first quotient digit, the product of this digit and the divisor is subtracted from the dividend. If an (incorrect) estimate of 9 is made in the above example, and 9 × 1944 is subtracted from 17495, the result is negative. The reason is that the estimate of 9 is too large. The next step is to restore the dividend to its original value and try again.

In binary, this process is simpler because the only possible estimates are 0 and 1. If 1 is too big, then 0 must be used. Another problem is determining where to begin; in other words, should we try to divide 1944 into 1 or 17 or 174 or 1749 or 17495 or 174959 or 1749598? The correct answer is 17495, but it is not easy for a computer to determine that. It simply tries them successively, generating leading zeros in the quotient, until one of the trial dividends is large enough.

Figure 4-19 depicts the division of 31416 by 25 in binary. First, attempts are made to divide 11001 into 1, 11, 111, and 1111, all of which are too small; that is, 11001 does not even "go into" them once. Finally, 11110 is tried. This number succeeds because 11001 "goes into" it once, with a remainder of 101. The next step is the famous "bring down the 1" step that has perplexed millions of little children. A

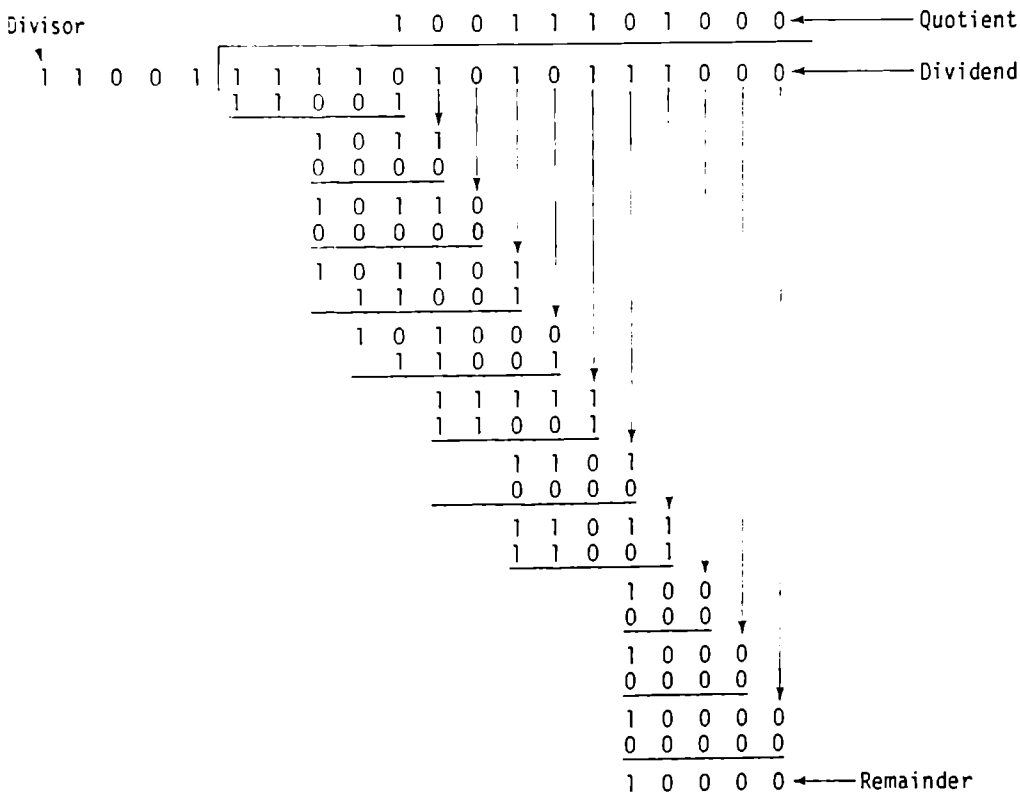


Fig. 4-19. Division of 31416 by 25 in binary.

better way of visualizing the situation is to think of a double-length accumulator, 32 bits in the case of our 16-bit target machine, containing the dividend as follows:

0000000000011110 1010111000000000

After subtracting 11001 from 11110, we get

0000000000000101 1010111000000000

Now, instead of "bringing down the 1," we left shift both 16-bit registers to yield

0000000000001011 0101110000000000

When we try subtracting 11001 from 1011, we get a negative result, which must be restored by adding 11001 back again. The quotient digit of 0 is recorded, and the accumulator is left shifted again. After discovering that 11001 does not "go into" 10110 either, another left shift gives 101101 as the trial dividend. This time, subtracting 11001 succeeds, yielding a quotient digit of 1. The process is repeated 16 times. At the end of the algorithm, the remainder can be found in the left half of the accumulator.

The microprogram statements beginning at DIV in Fig. 4-15 use this algorithm. First, the absolute values of the dividend and divisor are computed (the quotient is

	A	D	X
	0000 0000 0000 0000	0111 1010 1011 1000	0000 0000 0000 1111
	0000 0000 0000 0000	1111 0101 0111 0000	0000 0000 0000 1110
	0000 0000 0000 0001	1110 1010 1110 0000	0000 0000 0000 1101
	0000 0000 0000 0011	1101 0101 1100 0000	0000 0000 0000 1100
	0000 0000 0000 0111	1010 1011 1000 0000	0000 0000 0000 1011
	0000 0000 0000 1111	0101 0111 0000 0000	0000 0000 0000 1010
	0000 0000 0000 0101	1010 1110 0000 0001	0000 0000 0000 1001
	0000 0000 0000 1011	0101 1100 0000 0010	0000 0000 0000 1000
	0000 0000 0001 0110	1011 1000 0000 0100	0000 0000 0000 0111
	0000 0000 0001 0100	0111 0000 0000 1001	0000 0000 0000 0110
	0000 0000 0000 1111	1110 0000 0001 0011	0000 0000 0000 0101
	0000 0000 0000 0110	1100 0000 0010 0111	0000 0000 0000 0100
	0000 0000 0000 1101	1000 0000 0100 1110	0000 0000 0000 0011
	0000 0000 0000 0010	0000 0000 1001 1101	0000 0000 0000 0010
	0000 0000 0000 0100	0000 0001 0011 1010	0000 0000 0000 0001
	0000 0000 0000 1000	0000 0010 0111 0100	0000 0000 0000 0000
At DIVEND	0000 0000 0001 0000	0000 0100 1110 1000	0000 0000 0000 0000

Fig. 4-20. Values of A, D, and X just before the line labelled DIVLOOP in Fig. 4-15 is executed each time through the loop dividing 31416 by 25.

negated at the end if need be). The 32-bit accumulator consists of the A register (left half) and the D register (right half). The 2's complement of the absolute magnitude of the divisor is kept in the MBR to speed up the subtraction. The absolute magnitude of the divisor is kept in C to speed up restoring the trial dividend when it becomes negative. Figure 4-20 shows A, D, and X just before the DIVLOOP statement is executed each time through the loop for the division of 31614 by 25. Note that the top of the target machine stack is the divisor, not the dividend.

A few words about this division algorithm are in order. However complicated it may appear at first, this algorithm is one of the simplest in use for division of binary numbers. It is also one of the slowest. A big improvement would be to refrain from restoring A when it becomes negative (line 79) and instead add the divisor rather than subtract it on the next iteration. Such a method is called nonrestoring division (because it does not restore the accumulator). It saves the extra cycles needed for restoring the dividend. If you are interested in high-performance multiplication and division algorithms, consult the list of suggested readings under the heading "Binary Numbers and Arithmetic."

4.8.3. Perspective

The time seems appropriate to stop and reflect on what microprogramming is all about. The basic idea is to start with a very simple hardware machine (the microprogramming level of our example computer has only two simple instructions) and use it to write an interpreter for a more sophisticated target machine. This target machine can, in turn, serve as a host for yet another virtual machine, and so forth.

The target machine we have been using is admittedly very simple, for didactic purposes. An interpreter for the complete IBM 370 conventional machine level would be appreciably longer than the one shown in Fig. 4-15. Nevertheless, 370 interpreters do exist, and the technique of executing conventional machine language programs by interpretation has proven a good one through years of actual practice.

The organization of the microprogramming level of our example computer is also very simple. We now turn our attention to the organization and design of more sophisticated host machines.

4.9. DESIGN OF THE MICROPROGRAMMING LEVEL

In Chapter 3 it was shown why short instructions were better than long ones—namely, to save space in memory and to increase the number of instructions that can be fetched from memory per second. Precisely the same considerations apply at the microprogramming level. A microinstruction with one bit per gate is attractive because of its conceptual simplicity and ease of implementation (the 1's and 0's in the instruction can be applied directly to the gates), but a reasonable-sized computer may have a few hundred gates, making a microinstruction that is unacceptably long. In the following sections other designs will be examined.

4.9.1. Encoded Fields

Not all combinations of 40 bits are legal microinstructions for our machine. From Fig. 4-10 we see, for example, that gates 1, 2, 3, and 4 may not all be open simultaneously, for the adder may have only one left input. Similarly, although, in principle, there is no reason why gates 32, 33, 34, 35, 36, and 37 could not be open simultaneously, actually, there is rarely a need for more than one to be open at a given time.

The impossibility of having more than one left and one right input and the lack of need for more than one output suggest that the adder inputs and outputs could be specified in 10 bits instead of 23. Figure 4-10 has 10 left adder input possibilities, which requires 4 bits, 7 right adder inputs, which requires 3 bits, and 6 outputs, which also requires 3 bits. In a total of only 10 bits, 2 inputs and an output could be uniquely specified.

Such a format is called **encoded**, as opposed to **decoded**. If the number of possible inputs or outputs to a functional unit (such as an adder) is N, the saving in bits produced by encoding is approximately $N/\log_2 N$. For 32 inputs, only 5 bits are needed to select an input, yielding a savings of more than a factor of 6.

The disadvantages of encoding are twofold. First, for each encoded field in a microinstruction, a decoder (shown in Fig. 4-21) is needed for actual control of the gates. The saving in micromemory cost will probably be more than the cost of the decoders, however. Second, some flexibility is lost. On line 43 of Fig. 4-15, gates 33

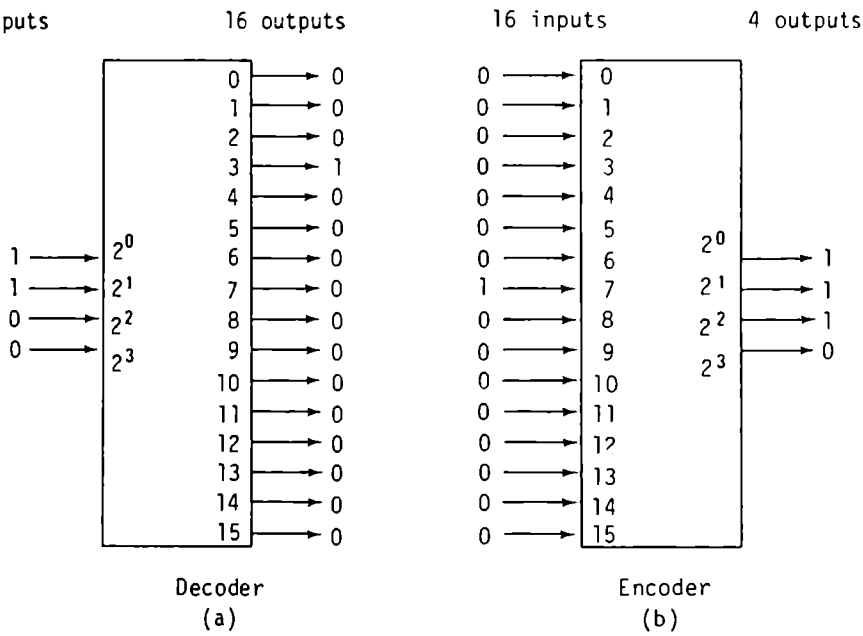


Fig. 4-21. (a) A decoder with four inputs can select any one of 16 output lines. (b) If exactly one of 16 possible input lines contains a 1, the number of this line can be encoded in a 4-bit number.

and 37 are both open during the same cycle. In the encoded form, this situation is impossible and an extra microinstruction is required, at a cost in both memory and speed. The designers of encoding schemes try as much as possible to group together into a field gates that are mutually exclusive; that is, no more than one can be on during a given cycle.

4.9.2. Horizontal versus Vertical Organization

The design proposed in the previous section in which GATE instructions contain a one-bit opcode, and three-address fields is a hybrid format in which gates 18 to 31, 38, and 39 still control individual gates. Each one of these instructions contains

1 bit for the opcode
4 bits for the left adder input
3 bits for the right adder input
3 bits for the adder output
10 bits for the individual data paths (gates 18 to 27)
4 bits for complementing and shifting (gates 28 to 31)
2 bits for memory read/write (gates 38 to 39)

making 27 in all, a saving of 13 bits from the first design, which required 40 bits.

Another area in which the design can be made less ad hoc is the area of data paths. The data paths shown in Fig. 4-10 are rather arbitrary and were chosen in such a way as to avoid unnecessary data paths, extra gates, and extra bits in the microinstruction. Figure 4-22 shows an alternative organization, in which the registers are connected via data buses instead of with separate connections for each data path. Because 15 registers is an awkward number, not being a power of 2, an extra scratch register, the E register, has been added, thus bringing the total to 16 registers.

Figure 4-23 shows a possible instruction set for the microprogramming level organization of Fig. 4-22. In this design, the 19-bit microinstructions contain a one-bit opcode field that selects one of two possibilities: GATE and TEST, as in the original example. The GATE instructions are broken down into seven fields: the opcode, left adder input, right adder input, adder output, complement control, shift control, and memory control. These GATE instructions are more powerful than the original ones because they allow several possibilities the original ones did not allow, such as using IR as adder input or output.

It is instructive to compare the two hypothetical architectures discussed. The one described by Figs. 4-10 and 4-13 has a long (40-bit) word and no encoding at all. Such a design, in which each microinstruction bit directly controls one gate, is called a **horizontal** design. The architecture of Figs. 4-22 and 4-23 has a shorter (19-bit) word and seven fields in its GATE microinstruction. Designs with many highly encoded fields in their microinstructions are said to be **vertical**. The microprogram-

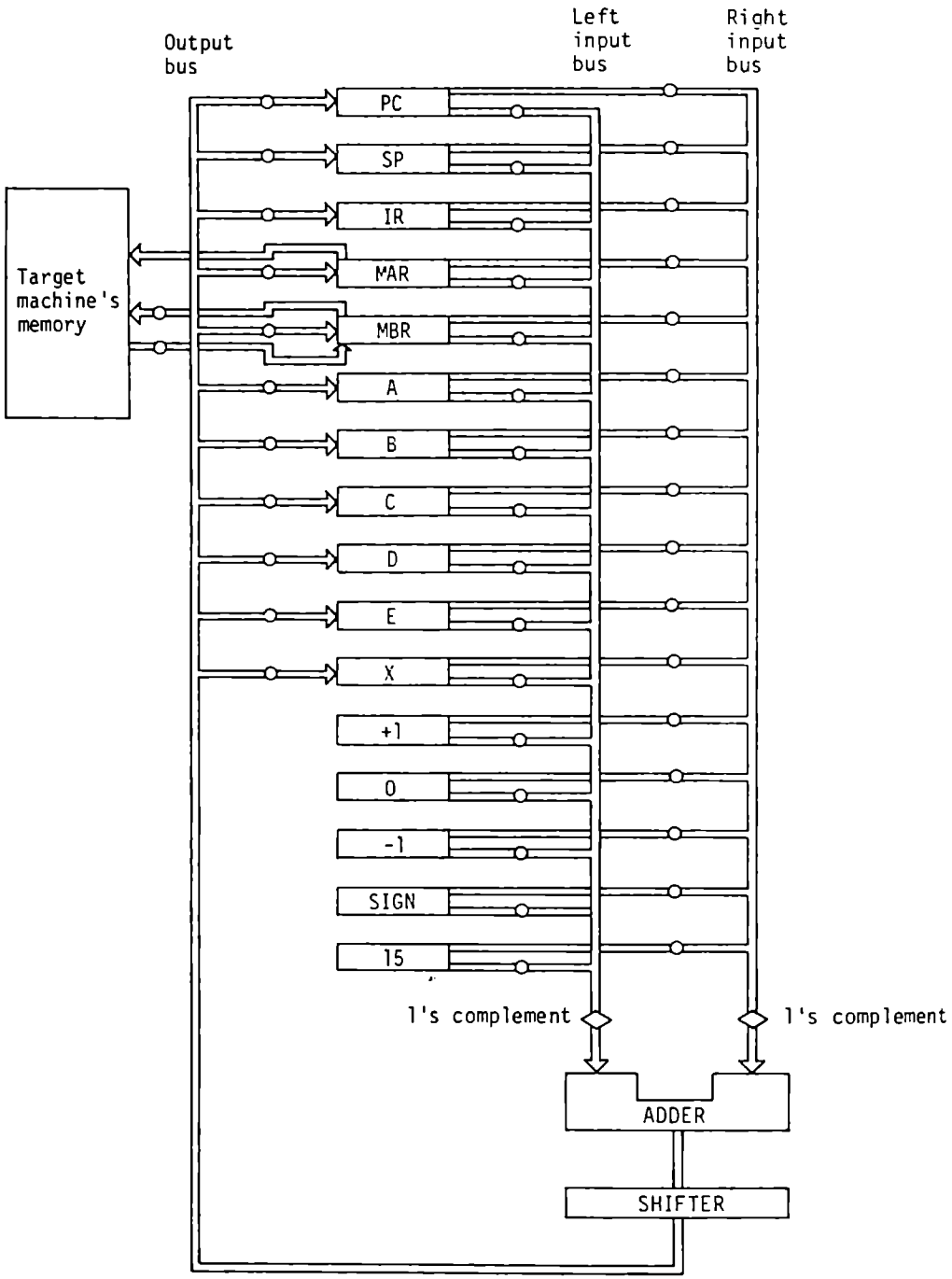
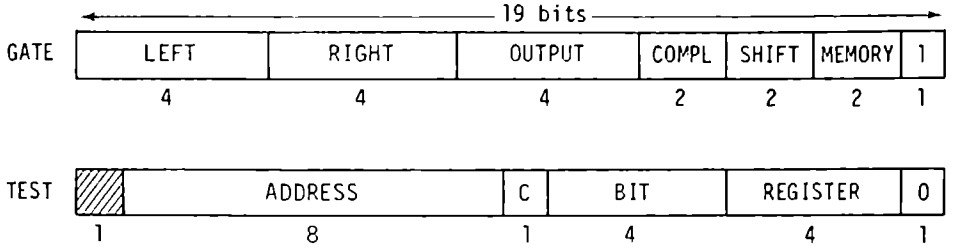


Fig. 4-22. The organization of a microprogramming level with three buses.

Microinstruction formats



Description of the fields

```
LEFT,RIGHT,OUTPUT and REGISTER  Select 1 of the 16 registers according to
   0 PC                4 MBR           8 D         12  0
   1 SP                5 A             9 E         13 -1
   2 IR                6 B            10 X         14 SIGN
   3 MAR               7 C            11 +1        15 15


COMPL     0 Do not complement either input
          1 Complement left input
          2 Complement right input
          3 Complement both inputs

SHIFT     0 Do not shift output
          1 Shift output left 1 bit
          2 Shift output right 1 bit
          3 Not used

MEMORY    0 No memory transfer during this cycle
          1 Read from memory
          2 Write into memory
          3 Not used

BIT       Selects bit to be tested (0 - 15)

C         0 Jump if tested bit is 0
          1 Jump if tested bit is 1

ADDRESS   An absolute micromemeory address to be jumped to
```

Fig. 4-23. Microinstructions used on the machine of Fig. 4-22.

ming level of some computers has 20 or more fields in its microinstructions (the PDP-11/40 microprogramming level has 26 separate fields, for example).

Horizontal and vertical organizations represent opposite ends of a spectrum of possibilities. The two machines described in this chapter differ in two major ways:

1. The horizontal design is highly parallel; the vertical design is not.
2. The horizontal machine's word is more than twice as long as the vertical machine's word.

Because the vertical organization allows only one function per cycle, tasks that take

only one microinstruction on the horizontal machine may take two or three on the vertical one. The results are slower execution and more microinstructions for the vertical machine. On the other hand, the smaller amount of memory needed to store the vertical microinstructions may very well offset these factors.

There is another difference between the horizontal and vertical designs, which is not illustrated in these examples. If a horizontal machine has N registers and is richly interconnected, meaning that there are data paths between most pairs of registers, then the number of connections and gates needed tends to grow as N^2. A machine with 64 registers, every one of which was connected to every other one, would need 4032 gates, which implies a microinstruction of at least 4032 bits to control them.

The most parallelism one could hope for would be 32 transfers per cycle, using 32 registers as sources and 32 as destinations. Thus the cost per transfer, in terms of the number of bits of memory required, increases as N increases. Worse yet, achieving an average of more than two or three transfers per cycle is highly unlikely, no matter how many registers are available.

By way of contrast, a microprogramming level organized around a small number of buses, such as that of Fig. 4-22 would only need 18 bits to specify a transfer on a 64-register machine, assuming a microinstruction format that consists of two input fields and one output field, all of six bits length. Consequently, machines with many registers are almost never purely horizontal. However, if a few data paths are heavily used—for example, PC to MAR—a special data path for that transfer may be worthwhile, even in an otherwise largely vertical machine.

4.9.3. Memory Cycle Ratios and Overlapped Execution

The examples described so far, have always included a tacit assumption that a main memory cycle is as short as or shorter than a processor cycle. Because the processor cycle time is strongly influenced, if not completely determined, by the micromemory cycle time, the ratio of main memory cycle time to micromemory cycle time is a critical design parameter.

This ratio will never be less than 1, which would mean that the target machine's memory was faster than the host machine's memory. Interpreting a target machine instruction usually requires only one or two main memory references, but it may require dozens of micromemory references. Consequently, the performance could be improved by keeping the target machine's program and data in the slower micromemory and the microprogram in the faster main memory, thereby changing the ratio.

A ratio of exactly 1 is not only possible but commonplace. This situation occurs if there is only one physical memory, used by both the target machine program and the microprogram, as on the 370/145.

In general, designers attempt to improve performance by making the micromemory as fast as possible. They do so by using an advanced technology for the micromemory and the standard technology for the main memory. As a result, micromemories

tend to be faster, more expensive, and thus smaller than main memories. On some machines, read-only micromemories are used to increase their speed.

The principal consequence of a high ratio of main memory cycle time to micromemory cycle time is that once the microprogram starts the main memory performing a READ operation, it must wait many cycles until the needed word is available in the MBR. This situation is illustrated in Fig. 4-24 for a ratio of 10: 1 and the PUSH instruction of Fig. 4-15, with time measured in processor cycles. Here the CPU is idle 79 percent of the time, waiting for the main memory to complete a cycle. As the ratio of cycle times increases, the utilization of the CPU drops.

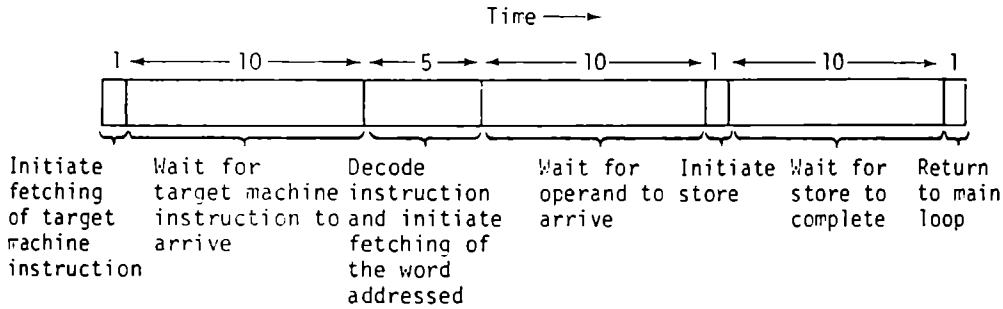


Fig. 4-24. Timing of the PUSH instruction of Fig. 4-15 in units of a microinstruction cycle, assuming a ratio of main memory cycle time to microinstruction time of 10 to 1. The shaded areas show when the microprogram is waiting for the main memory to complete a cycle.

Economic pressures generally dictate that high idle times of expensive gadgets, such as computers, are unacceptable. One approach to this problem is to use a faster (hence more expensive) main memory. Another approach is to overlap target machine execution. As an example, consider the interpretation of multiplication instructions by the interpreter of Fig. 4-15. In the best case (0 times 0), the interpretation of a multiplication instruction will take 142 microinstructions, exclusive of the main memory references. That might be enough time to fetch the next 10 or 12 target machine instructions in advance, so that they will be available when needed. This strategy is often called **prefetching**, or **look ahead**.

Fetching many instructions in advance requires that there be a place to keep them. The microprogramming level could easily be provided with more scratch registers, connected the same way as B and C in Fig. 4-10. However, using additional scratch registers increases the number of gates, and the number of bits in the microinstruction, as well as the width and cost of the micromemory. Furthermore, fetching only instructions is not enough, because the microprogram will then spend most of its time waiting for data.

When interpreting a stack-oriented target machine, it is to be expected that many memory references will be for fetching data from the stack. By keeping the top few words of the stack in internal scratch registers, these reads and stores can be elimi-

nated. Similarly, the microprogramming level for machines like the 370 and PDP-11 are commonly designed with enough scratch registers to hold all the target machine registers, although some of the smaller 360 models used main memory for them, in order to keep the cost down.

Another approach is to write the microprogram in such a way as to overlap execution of target machine instructions. In other words, the interpretation of a target machine instruction could be started, and perhaps even finished, before its predecessor had been completed. To do so requires a highly sophisticated algorithm for determining the next step. Such an algorithm must attempt to have instructions and operands arrive from main memory at a rate approximately equal to the rate the CPU can utilize them, which, in turn, is highly dependent on the instructions themselves. Moreover, certain interlock problems mentioned below must be solved.

As an example of the kinds of interlock problems that can occur, consider the evaluation of

```
L = I + J × K
```

by

```
PUSH I
PUSH J
PUSH K
MUL
ADD
POP L
```

If the addition is started before the multiplication is completed, it will use the wrong operands. Care must be taken that no instruction depending on previous results is started before those results are available.

Conditional jump target machine instructions also pose problems. When a conditional jump is encountered while previous instructions are still being interpreted, it may not be possible to determine if the condition will be met or not. The microprogram is then faced with a choice of whether to continue the prefetching from the instruction following the conditional jump or from the jump address. It could follow both paths and discard the results of the incorrect one when the condition is finally determined, but ensuring that no irreversible changes occur meanwhile is more than a little tricky. Among other things, if any assignments were performed on the wrong path, the variables changed must be restored to their previous values. The potential need to undo assignments implies that the previous values of the variables modified must be saved somewhere. Moreover, if the condition involved a time-consuming calculation, such as the PL/I statement

```
IF A × B × C × D × E < 1000 THEN CALL HOME;
```

many more conditional jumps might be encountered before the condition was finally determined. The microprogram could easily find itself working on five or ten different instruction streams at once, the undoing of all but one of which could be a nightmare.

Alternatively, the microprogram could make a guess based on past experience

("this fellow usually fails the tests and does not jump") and only follow the chosen path, with the intention of undoing all results if the guess later proves to be bad. In this approach, the speed of the target machine depends on how good the microprogram's guesses are.

The University of Manchester MU5 computer uses an interesting technique to improve the accuracy of the guesses (Sumner, 1974). The MU5 keeps track of the most recent conditional jump instructions executed, and whether or not each one succeeded. Because many conditional jumps are related to looping, they tend to repeat their success or failure many times in a row.

4.9.4. Nanomemories

The designs discussed so far have all had two memories: the main memory—that is, the target machine's memory—and the micromemory, where the microprogram is stored. A third memory, or nanomemory, makes interesting tradeoffs between horizontal and vertical microprogramming possible. The basis for this design is that in the horizontal, one-bit-per-gate approach, only a few combinations of the gates are ever used. However, they are used repeatedly. In Fig. 4-15, for example, 9 of the 12 target machine instructions need to pop a word off the stack.

The nanomemory has a long word length, one bit per gate, but the total number of words in the nanomemory is small, perhaps 64 to 256. Each nanomemory word specifies the status of all the gates in the machine during one cycle. The microprogram consists of a series of nanomemory addresses. The microinstructions no longer directly control the gates, but instead point to nanomemory words that do control the gates. Thus, if two, or three, or more microinstructions require the same gate control bits, they can all point to the same nanomemory location. If the machine has many gates but the same combinations occur frequently, the scheme is an attractive one, because it provides the possibility of parallel data flow and simple hardware, without the disadvantage of a large micromemory.

Figure 4-25 illustrates the concept of a nanomemory. In part (a), a microprogram of 4096 microinstructions for a 200-gate horizontal microprocessor is shown. A total of 819,200 bits of micromemory is needed to store the 4K microprogram. Suppose that a careful study of the microprogram revealed that only 64 different microinstructions of the 2^{200} possibilities were actually used. A special 64-word, 200-bit-wide nanomemory could be used to store these 64 microinstructions. Each microinstruction in the original program could be replaced by the address of the nanomemory word containing that microinstruction. Because there are only 64 words in the nanomemory, the micromemory need only be six bits wide, as shown in Fig. 4-25(b).

The microprogram is executed as follows. The first six-bit word is fetched from the micromemory. It is then used to select a nanomemory word, which is fetched and placed in a gate control register. The bits of this register are then used to open gates for one cycle. At the end of the cycle, the next six-bit word is fetched from the micromemory and the process is repeated.

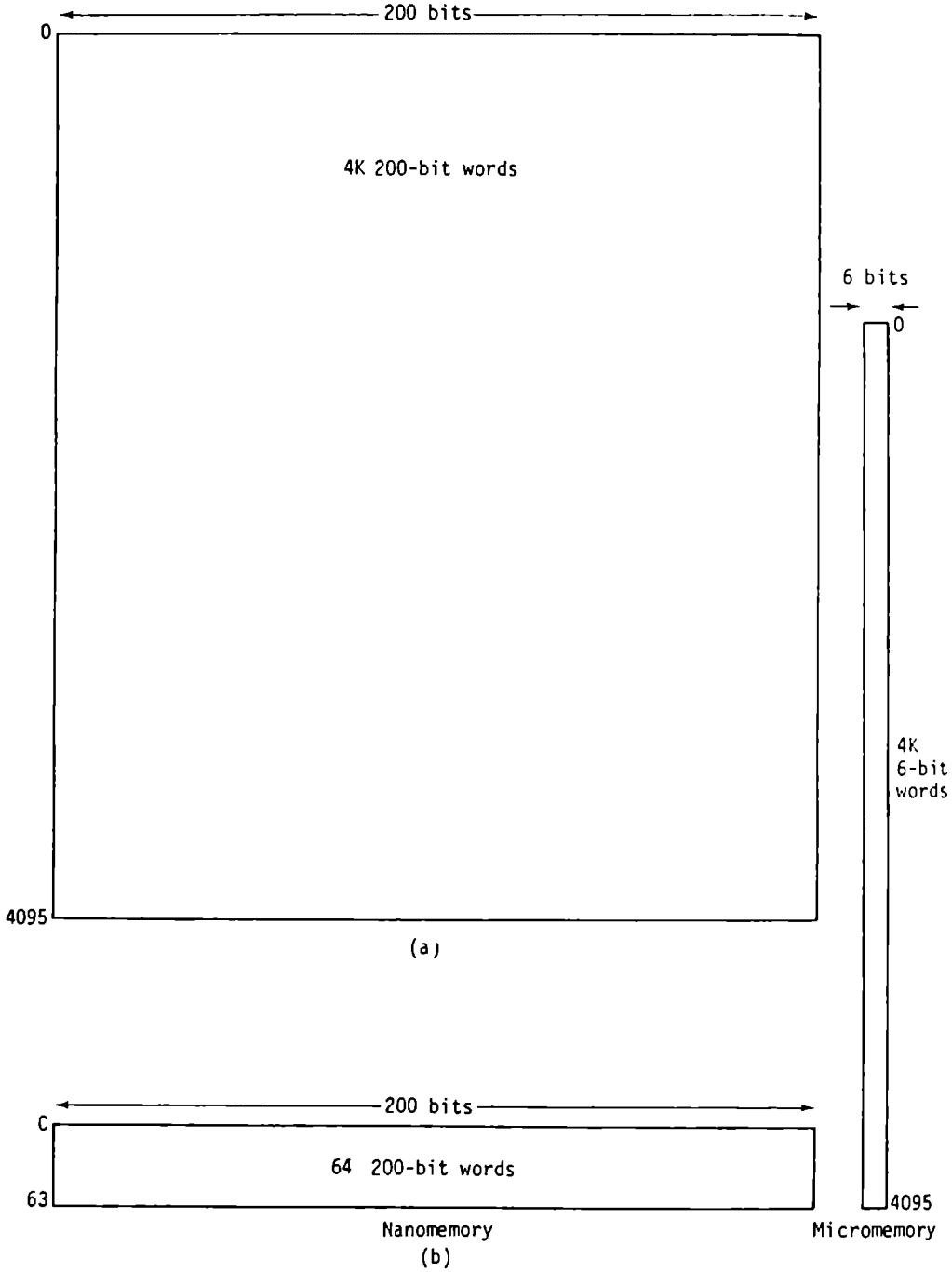


Fig. 4-25. (a) Micromemory of 4K 200-bit words. (b) Nanomemory of 64 200-bit words and micromemory of 4K 6-bit words.

In this example, the nanomemory requires 12,800 bits, and the six-bit-wide micromemory requires 24,576 bits, a total of 37,376 bits, compared to 819,200 bits in Fig. 4-25(a). The nanomemory has retained the principal advantage of horizontal microprogramming—parallel transfers—while at the same time reducing the memory requirements by 95 percent. It does have the disadvantage of increasing the basic cycle time of the CPU to the sum of the micromemory's and nanomemory's.

In order for the nanomemory idea to be of value, a situation in which microinstructions are repeated over and over in the microprogram is needed. If all 4096 microinstructions in Fig. 4-25(a) were different, the idea fails completely. In this regard, it is useful to note that microprocessor TEST-type instructions specifying a relative rather than an absolute address are very desirable. For example, instructions of the form

```
IF BIT(15,MBR) = 1 THEN SKIP 4 MICROINSTRUCTIONS;
```

are far more likely to be used repeatedly than instructions of the form

```
IF BIT(15,MBR) THEN GO TO L3917;
```

4.9.5. Universal versus Specific Microprogramming Levels

Another design issue is whether the microprogramming level is intended to interpret one or more target machines, or a large number of widely differing target machines. The microprogramming level architecture used in the 370 series is specifically designed to maximize performance for a target machine consisting of the 370 conventional machine level and on some models, the IBM 1401 or IBM 7094 conventional machine levels as well.

The 370 CPUs were never intended to run microprograms that interpret Cyber 70 programs nor are they especially well-suited for this task. Among other difficulties, the internal registers are all too short, the memory bus widths are not appropriate for fetching units of 60 bits, and the address decoding logic is different.

On the other hand, the Burroughs B1700 was designed with a general-purpose microprogramming level. It can interpret 370, Cyber 70, PDP-11, or any other conventional (or unconventional) machine language program without great difficulty. Because of its generality, it is perhaps not as efficient at interpreting 370 programs as the IBM CPUs, which were carefully tailored to that purpose.

An interesting problem is how to design a microprogramming level in such a way that it can conveniently handle several target machines, each having a different word length. One way to do so is as follows.

Each bit in the target machine memory has an address. In other words, the cell size is one bit. Although the target machine may conceive of the memory differently—for example, eight bits per cell. To implement this, when an address is put in the one-bit-per-cell MAR and a read is requested, 64 bits are fetched, beginning at the specified bit, and put into the corresponding, 64-bit-wide MBR. Assuming that the word length of the target machine is 64 bits or less, the MBR will contain a word, plus pieces of other words.

The functional units, data paths, and registers are all 64 bits wide. The functional unit contains a 64-bit wide mask register, loadable by the microprogram. All operands and results are automatically AND'ed with the mask. To handle a 27-bit target machine, the mask would contain 27 ones and 37 zeros. The boolean AND of this mask register with the operands eliminates the unwanted bits and thereby performs operations on the desired word length. Similarly, stores back into memory use another mask to ensure that only the desired number of bits are stored and that adjacent words are not disturbed. To handle word lengths greater than 64 bits, the microprogram must perform the operations in two or more parts.

A less expensive alternative to having 64-bit data paths, and hoping that 64 bits will suffice, is to have a smaller MBR (24 bits on the B1700) and a flexible method for extracting bits from the middle of registers. Another useful feature is the ability to fetch variable-sized bit fields from memory, up to a maximum of the MBR width. The B1700 uses this approach.

4.9.6. Review of Microprogramming Level Organization

Frequently, the most difficult part of getting acquainted with a new computer is trying to organize all the facts available about it. In this section we list some questions you should ask about the microprogramming level of any computer you encounter.

Registers

1. How many registers are there?
2. How many bits does each register contain?
3. Are any of the registers divided into separately addressable subregisters?
4. Do any registers have specific functions (e.g., microprogram counter)?
5. How many MARs and MBRs are there and what are their widths?
6. Which registers hold target machine registers?
7. Which registers are available for microprogram working storage?

Data Paths

1. Which registers have direct connections to other registers?
2. How many buses are there and what are their widths?
3. Which registers can be gated onto each bus?
4. Are there any "hidden" registers with no accessible data paths?

Memories

1. How many memories are there?
2. How many bits in the basic addressable unit of each memory?

3. How many addressable units (cells) does each memory have?
4. Which MARs and MBRs operate each memory?
5. Can the number of bits transferred per cycle be set by the microprogram?

Microinstructions

1. How many microinstructions are there?
2. What are the microinstruction formats?
3. Are the microinstructions predominantly horizontal or vertical?
4. How much parallelism do the microinstructions allow?
5. How is i/o accomplished?
6. What does each microinstruction do?

Functional Units

1. Can the ALU add 1's and 2's complement binary numbers and also decimal numbers?
2. Can the ALU subtract?
3. Are there special hardware facilities for multiplication and division?
4. Are there hardware floating-point arithmetic units?
5. What kinds of shift and rotate operations are possible?
6. Is there a unit that performs boolean functions (AND, OR, etc.)?

Microinstruction Sequencing

1. How is the sequencing of microinstructions determined?
2. Does each microinstruction contain the explicit address of its successor?
3. Does each microinstruction contain one or more fields used to compute its successor?
4. Is there a conditional jump or skip microinstruction?
5. What is the range of addresses to which the conditional and unconditional jumps can branch?
6. Is there a method for microprograms to call microprocedures, saving the microprogram counter?
7. Can microprograms be interrupted (by i/o, for example)?

Overlap

1. What is the ratio of main memory cycle time to micromemory cycle time?
2. Can the CPU proceed while a main memory operation is taking place?

3. Are microinstructions prefetched before they are actually needed?
4. Are there mechanisms for facilitating overlapped target-machine-instruction interpretation?

4.10. ADVANTAGES AND DISADVANTAGES OF MICROPROGRAMMING

The ability for users to write their own microprograms (i.e., design their own target machines) has many advantages over conventional "fixed-design" machines.

First, the user can tailor the machine to the specific needs of his application, thus producing a target machine that is not only efficient but also conceptually simple for his application. In this way, the gap between the application and the machine can be narrowed. This approach is the basic idea behind constructing multilevel systems. The actual hardware can be awkward to program, but the machine that the applications programmer sees is far more sophisticated and appropriate to his problem.

Over the past decade or so, an enormous amount of effort has gone into preserving the conventional machine level as a distinct variety of virtual machine. This situation seems to be as much a result of ingrained habits as of carefully thought out strategy. Today most people program in problem-oriented languages, such as ALGOL 68, FORTRAN, PASCAL, PL/I, and COBOL. These programs are then compiled into what we have called level 3 machine language (which consists largely of conventional machine language instructions with certain additions). Finally, the level 3 programs are interpreted.

The realization that nearly all programs are first translated to an intermediate level, and that these intermediate level programs are then interpreted leads to an interesting and perhaps embarrassing question: "Is the conventional machine level the optimal interface between the compiler and the interpreter?" There is little reason to believe that it is. A common situation is for the code generated by a compiler to use fewer than 20 percent of the available instructions. Consequently, the users of problem-oriented languages are paying (indirectly) for the micromemory and hardware required to interpret instructions that their compiler never generates. A microprogrammable computer holds out the hope of designing a better interface. A fixed-design machine excludes that possibility. This subject is discussed in more detail in Section 7.5.

Second, the user may write programs whose procedures run on another machine. The main program may be written in the machine language for the brand X machine, and some of the procedures may be written in the machine language for the brand Y machine. Doing so is easy if, in the procedure call instruction, there are a few bits that specify the microprogram to be used for the called procedure. The microprogram would examine those bits and jump to the appropriate section of the interpreter.

Changing machines would require no more effort than the execution of a jump

microinstruction. The ability to mix machines would be especially common if the programmer wanted to use procedures he wrote years ago for a different computer, or if he wanted to use procedures written at another computer center, which had a different computer, or if he simply wanted to take advantage of a large, existing procedure library for another machine.

Third, microprogramming can extend the useful life of a computer. When new ideas about machine architecture or instruction sets are developed, the target machine(s) in use can be modified to include them by simply changing their microprograms. The alternative of throwing out the old machine and buying a new one is not only wasteful, but the transition is fraught with problems as well.

Fourth, when the inevitable transition to a new machine arrives, microprogramming can make it possible for users to continue using their old software by simulating the old conventional machine level on the new hardware. This technique may not be as efficient as rewriting everything for the new machine, but it does save a lot of work because all that is needed is a microprogram to interpret the old conventional machine level.

Fifth, microprogrammed computers are easier to design and are conceptually simpler than hard-wired ones. Consequently, machine design becomes accessible to more people, and less specialized training is required to design a machine.

Sixth, maintenance is easier and better. A clean and simple design means less hardware, hence fewer items to break down. Unlike hardware, if a microprogram works today, it will also work tomorrow. Since the hardware is simpler, training repairmen is easier. Furthermore, the microsoftware itself can frequently check for errors and pinpoint defective components. In hard-wired machines, this is more difficult to do.

If the microdiagnostic procedure discovers that the floating-point add unit is defective, it could load a microprocedure from secondary memory to interpret the floating-point ADD instruction until the ailing hardware has been repaired. The result is a lower performance system, but that is certainly preferable to one that does not perform at all.

The principal disadvantage of microprogrammed processors lies in the area of cost/performance. If a certain performance machine is required, it may be cheaper to construct it from hardware than to interpret it in software. The reason is that, typically, 10 microinstructions are required to execute one target machine instruction. As a result, in order to achieve a target machine with an instruction time of N nanoseconds, a CPU with a microinstruction time of N/10 nanoseconds will be required.

If an N nanosecond instruction time is near the limit of the existing technology, an N/10 nanosecond CPU may be too expensive to be acceptable. However, the cost of a hardware processor of a given power has been decreasing by a factor of 10 per decade over the past few decades; and if this trend continues, objections of the form "that is a fine idea, but it is too expensive" are apt to become less important.

4.11. THE IBM 370/125's MICROPROGRAMMING LEVEL

In the following sections we will examine the microprogramming level of three commercially available computers: the IBM 370/125, the PDP-11/40, and the Burroughs B1700. The IBM 370 model 125 is one of the smaller models of the 370 series, and it is widely used for business and administrative applications. A 370/125 consists of not just one processor, but of at least three processors, and possibly more, depending on the number of i/o channels.

Each 370/125 has one processor that serves as the CPU, fetching, examining, and executing 370 programs stored in main storage. The IBM designation for the processors used as CPUs on the 370 series is the model number plus 3000: the IBM 3125 processor is used as the CPU on the model 125. Each 370/125 also has a processor called the "service processor" or SVP. The SVP has three basic functions:

1. Loading microprograms into itself and all the other processors from a special disk drive.
2. Handling communication with the console operator.
3. Detecting hardware malfunctions in the other processors and keeping a record of them for the maintenence personnel.

Every 370/125 also has one or more processors called IOPs that perform all the i/o for the 370/125. Because the 370/125 is commonly used in applications requiring substantial i/o, the machine may be equipped with several IOPs. In addition to these three kinds of processors, the 370/125 has a hard-wired (i.e., not microprogrammed) unit called the main storage controller (MSC), which coordinates all requests from the CPU, SVP, and IOPs for reading or writing main storage.

4.11.1. Architecture of the IBM 370/125 Microprogramming Level

Figure 4-26 is a simplified diagram of the components and data paths in the CPU, as well as portions of the main storage controller. Figure 4-27 summarizes the properties of these memories, registers, and functional units. The TDR, CDR, ALU inputs and outputs, shift unit, and CPU local storage are all 16 bits wide. Because the 370 conventional machine level general registers and data paths are 32 bits wide, the 370 general registers must be stored in two pieces and operations carried out separately on each piece. Thus the 3125 is both slower and cheaper than the 3145, which has 32-bit registers and data paths.

Four separate memories are shown in Fig. 4-26, each with its own purpose. The main storage, micromemory, and CPU local storage hold the 370 program, microprogram, and microprogram variables (including the conventional machine level's registers), respectively. The main storage controller also has a small memory of its own, to which the CPU, SVP, and IOPs have access. Each processor has an area

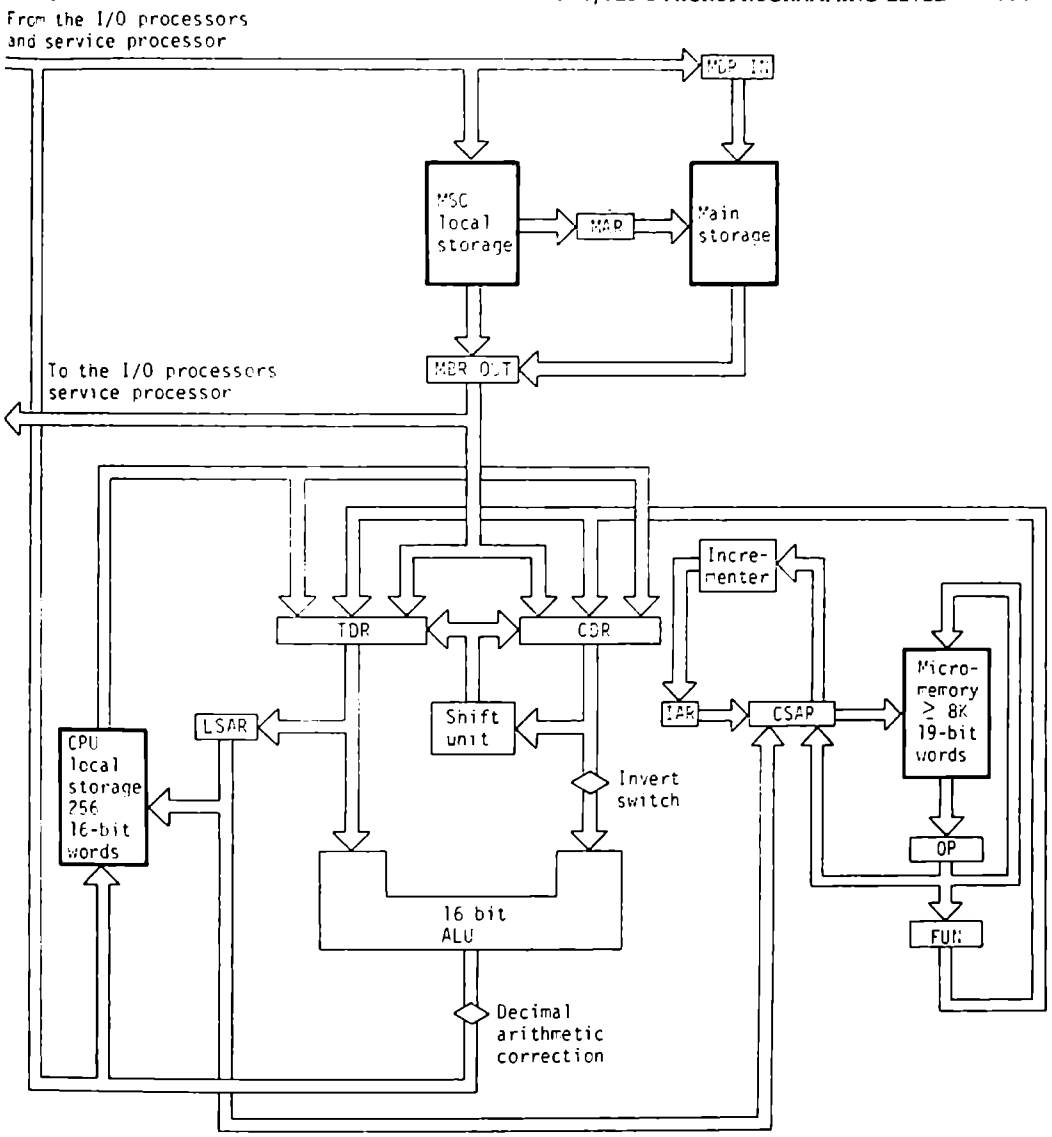


Fig. 4-26. A simplified diagram of the architecture of the IBM 3125's microprogramming level.

of this memory assigned to it. The 3125 microinstructions reference main memory by specifying one of four possible words within the MSC local storage. The selected MSC word is used as the main storage address and gated into the MAR. This addressing method requires only two bits in the microinstruction for main memory addressing. The four words are

The 370 level 2 IAR (program counter), used to fetch target level instructions.

MEMORIES

Name	Use	Cell size	Number of cells
Main storage	Holds the IBM 370 level 2 programs and data; this is the only memory the level 2 programmer is aware of	8	98K-128K
MSC local storage	Holds the program counter and other main storage addresses	23	32
CPU local storage	Holds the microprogram variables and the level 2 machine's registers	16	160-192
Micromemory	Holds the microprogram	19	8K-20K

REGISTERS

Name	Width	Use
TDR	16	Left input to ALU (true data register)
CDR	16	Right input to ALU (complement data register)
MBR IN	16	Halfwords to be stored in main storage are loaded here
MBR OUT	16	Halfwords read from main storage arrive here
CSAR	15	Micromemory MAR (control storage address register)
OP	19	Micromemory MBR (for reading and writing)
FUN	19	Holds microinstruction while it is being executed
LSAR	4 x 4	Used to address CPU local storage (local storage address registers)
IAR	3 x 15	Micro program counters (IAR0, IAR1, IAR2)

Notes: LSAR is 4 registers of 4 bits each
IAR is 3 register of 15 bits each

FUNCTIONAL UNITS

Name	Output width	Use
ALU	16	Binary addition, AND, OR, and EXCLUSIVE OR
Shift unit	16	Shifts 0-15 bits left or right with zero fill
Incrementer	15	Increments current micro program counter (IAR) by 1
Invert switch	16	Controls ALU right input: CDR, 1's complement of CDR, 16 ones or 16 zeroes
Decimal arithmetic correction	16	Performs correction needed to convert binary addition to decimal addition

Fig. 4-27. Summary of the components of the IBM 3125's microprogramming level.

Two operand addresses (target instructions can have, at most, two operands in memory).

A communication word common to all the processors, which is used primarily for passing i/o instructions from the CPU to the IOPs for execution.

Associated with each operand address is a length field, giving the length of the operand.

The box labeled LSAR consists of four registers of four bits each, rather than a single 16-bit register. The four registers, LSAR0, LSAR1, LSAR2, and LSAR3, can be used to form the eight-bit addresses needed to address the CPU local storage (locations 160 to 191 are optional, and locations 192 to 255 are reserved for possible future expansion). CPU local storage addresses can be formed in three ways:

1. All eight bits are contained within the microinstruction (direct addressing).
2. The leftmost four bits are contained within the microinstruction, and the rightmost four bits are taken from one of the LSARs.
3. The leftmost four bits come from LSAR0, and the rightmost four bits come from any one of the LSARs.

The first form is useful for accessing variables, counters, and so on. The second and third forms are useful for accessing the 370's level 2 registers. The high-order parts of the 16 general registers are kept at CPU local storage addresses 40 to 4F (hexadecimal), and the low-order parts are kept at addresses 50 to 5F. Access to the level 2 registers is easily performed by first loading the register number, 0 to F (hexadecimal), into an LSAR. CPU local storage addresses are then formed from the hex digits 4 or 5 (contained within the microinstruction itself), concatenated with the register number (from an LSAR).

There is a one-to-one correspondence between the 16 LSAR bits and the 16 TDR bits. Bits in the microinstructions enable gating of the TDR into the four LSARs. In this way, the register numbers can be loaded into the LSARs.

The 3125 has three microprogram counters (labeled IAR in Fig. 4-26). At any instant, one of them is the "current" one and is used for fetching microinstructions from the micromemory. When the main (micro) program is running, IAR0 is used. When the microprogram calls a (micro) procedure, the return address is put in IAR0, the procedure's starting address is put in IAR1, and IAR1 becomes the current microprogram counter.

A return from a procedure simply makes IAR0 the current microprogram counter again. In essence, IAR0 and IAR1 form a two-word stack. A procedure in the microprogram may not call another procedure. IAR2 is loaded and becomes current only when an interrupt occurs. The incrementer adds 1 to the current IAR after each microinstruction is fetched.

The invert switch controls the right ALU input. Its four positions correspond to

1. CDR
2. 1's complement of CDR
3. 16 one bits (−1 in 2's complement)
4. 16 zero bits

Once set by a microinstruction, it remains set until explicitly changed by another microinstruction.

4.11.2. IBM 3125 Microinstructions

The IBM 3125 microinstructions are neither purely horizontal nor purely vertical but a mixture of both organizations. An expanding opcode is used, with opcodes ranging from two to six bits. The remainder of the 19-bit microinstructions are divided up into fields whose number, size, and function differ for each of the 13 microinstructions. Furthermore, some microinstructions have several minor variations, with the actual format being determined by certain bits within the microinstruction.

Figure 4-28 lists the primary function of each microinstruction. In addition, some microinstructions have fields for performing additional functions, which may be unrelated to the primary function. They include

1. Return from procedure.
2. Set invert switch for use in subsequent ALU operations.

```
Load TDR or CDR

  1. Load 1 byte of TDR or CDR with immediate data. Zero or retain other byte.
  2. Load 16-bit halfword into TDR or CDR from CPU local storage.
  3. Load byte or halfword into TDR or CDR from main storage.
  4. Load leftmost 7 bits or rightmost 16 bits of MSC local storage in TDR
     or CDR.

ALU operation (ADD, AND, OR, or EXCLUSIVE OR)

  5. ALU operation. Store result in CPU local storage.
  6. ALU operation. Store result in main storage.
  7. ALU operation. Store result in MSC local storage.

Test and conditional branch

  8. Test for presence or absence of one out of 48 conditions.
  9. Branch on test. Conditional branch on outcome of previous test.
 10. Conditional branch on 1 of 10 conditions irrespective of previous test.

Miscellaneous

 11. Shift CDR left or right 0-15 bits with zero fill. Store result in CDR
     or TDR.
 12. Sense and control. Read or set timers, clocks and other system devices.
 13. Table look up. Used to examine level 2 machine opcodes and branch to
     microprogram address to start interpretation of the instruction.
```

Fig. 4-28. Principal functions of the IBM 3125 microinstructions. Each microinstruction has several secondary functions which are not shown.

3. Copy left byte and/or right byte of TDR into corresponding LSARs.
4. Increment or decrement an MSC local storage address by one or two bytes.

Other fields provide addresses and other information needed to perform the primary function, such as

1. ALU function (ADD, AND, OR, EXCLUSIVE OR).
2. Destination register for output from local or main storage (CDR or TDR).
3. Specification of CPU local storage addresses (immediate bits and LSAR numbers).
4. Determination of ALU output width (8 or 16 bits).
5. Immediate data (for loading either byte of CDR or TDR).
6. Enable or disable decimal arithmetic correction.
7. Handle carry bits for multiple-precision arithmetic.
8. Handle condition codes for multiple-precision arithmetic.
9. Specification of condition to be tested.
10. Branch address (partial or complete).
11. Enable or disable virtual memory (see Section 5.5).

Of course, not every one of the preceding 15 fields is present in every microinstruction.

The TEST microinstruction can test for the presence or absence of 48 different conditions or bits. They include the 16 TDR bits, various bits in the LSARs, certain hardware errors, certain exceptional conditions, such as floating-point overflow, interrupt signals from the other processors, the ALU carry bit, and the condition code.

The BRANCH ON TEST microinstruction uses the result of the most recent TEST to branch to a new microprogram location (14-address bits are provided). In addition, a two-bit field provides for optional procedure calling or returning (i.e., switching IARs).

In contrast, the CONDITIONAL BRANCH microinstruction can specify any one of 10 conditions to be both tested and branched upon in a single microinstruction. These conditions include specific LSAR and TDR bits, ALU output zero or nonzero, and ALU carry bit set.

The SENSE AND CONTROL microinstruction can read the status of various timers, clocks, and exceptional conditions into CDR or TDR, and can also set these devices and registers from CDR or TDR.

As an example of a microinstruction, the meanings of the fields in LOAD BYTE OR HALFWORD INTO TDR OR CDR FROM MAIN STORAGE are given below. The bits here are numbered from 0 (leftmost) to 18 (rightmost).

Bits	Meaning
0–4	The opcode for this microinstruction is 11111 (binary)
5	1 = leave procedure
6	0 = read into CDR; 1 = read into TDR
7	1 = copy (new) TDR bits 0 to 3 to LSAR0 and TDR bits 4 to 7 to LSAR1
8	1 = copy (new) TDR bits 8 to 11 to LSAR2 and TDR bits 12 to 15 to LSAR3
9–10	set invert switch: 0 = 1's complement; 1 = no change; 2 = all 1's; 3 = all 0's
11	1 = check for exceptional conditions and trap if present
12	0 = read byte (8 bits); 1 = read half word (16 bits)
13	1 = increment selected MSC local storage register by byte or half word
14	1 = decrement selected MSC local storage register by byte or half word
15–16	MSC local storage address: 0 = IAR; 1 = operand 1; 2 = operand 2; 3 = i/o common register
17	0 = update MSC address; 1 = update MSC length
18	0 = disable virtual memory address translation; 1 = enable it

The time required to execute a microinstruction is 480 nsec. The execution time of a 370 conventional machine level instruction depends on the number of microinstructions needed. A few sample 370/125 instruction execution times (in nanoseconds) follow:

LR	3840	(8 microinstructions)
AR	4800	(10 microinstructions)
L	6720	(14 microinstructions)
ST	7200	(15 microinstructions)
MR	123360 + 2880N	(2570 + 6N microinstructions), where N = number of one bits in the multiplier

4.12. THE PDP-11/40 MICROPROGRAMMING LEVEL

Like the 370 series, no two models of the PDP-11 series have the same architecture at the microprogramming level. In this section we will briefly describe the PDP-11/40 microprogramming level and compare it with the IBM 3125 microprogramming level described in Section 4.11.

4.12.1. Architecture of the PDP-11/40 Microprogramming Level

Figure 4-29 is a simplified diagram of the components and data paths in the PDP-11/40 CPU. Figure 4-30 summarizes the properties of these components. The machine is essentially divided into two parts: the arithmetic section and the control section.

The arithmetic section is the part on the left, and it includes the CPU local storage, the IR, PS, D, and BA registers, three multiplexers, the ALU, and the shift unit. The arithmetic section data paths are all 16 bits wide, except for the 18-bit bus address data paths. The arithmetic section is responsible for carrying out the execution of level 2 instructions.

The control section is the part on the right, and it includes the micromemory, the

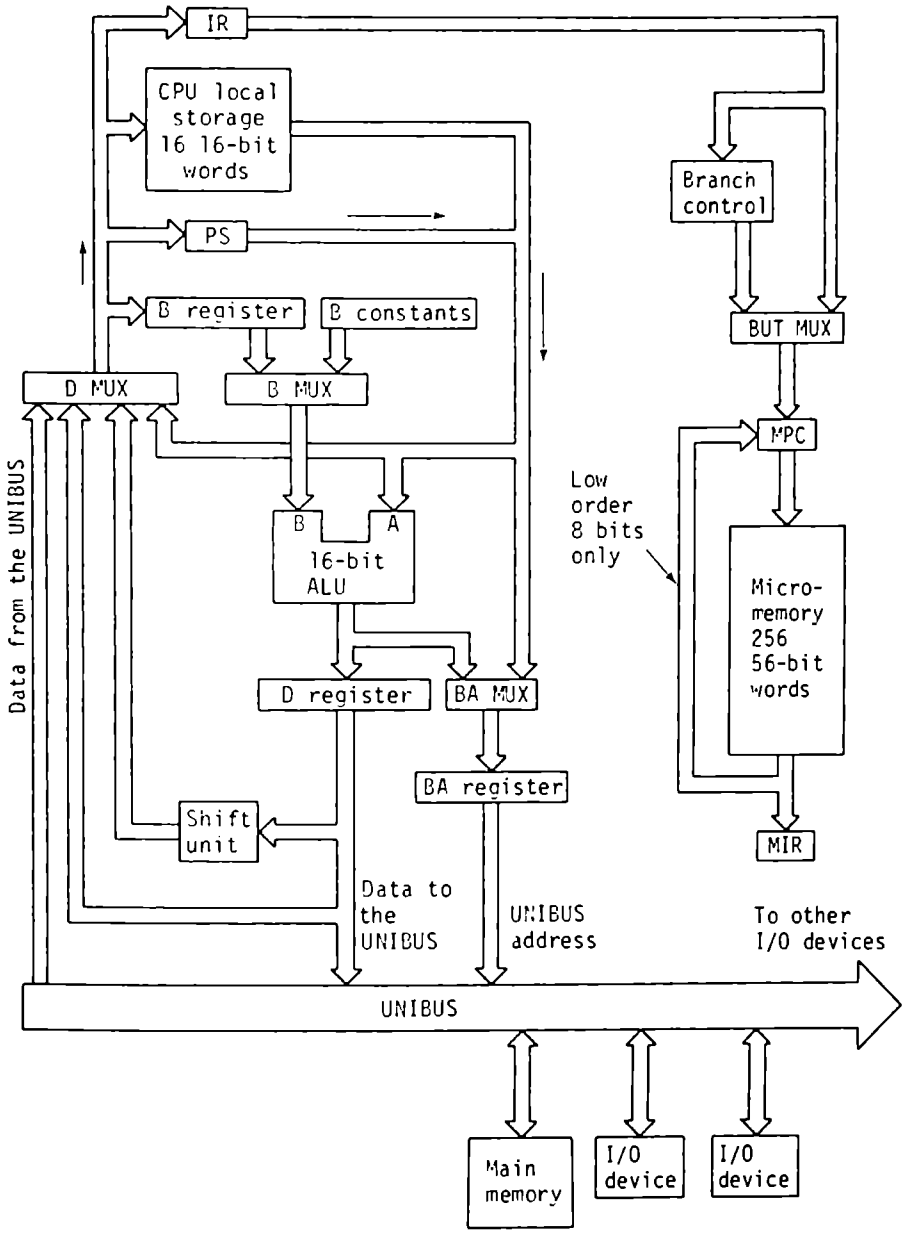


Fig. 4-29. A simplified diagram of the architecture of the PDP-11/40's microgramming level.

microprogram counter (MPC), the microinstruction register (MIR), the microbranch multiplexer (BUT MUX) and control. The control section is responsible for fetching the microinstructions from the micromemory and using the various fields in them to open and close gates within the arithmetic section. Although the micromemory is 56 bits wide, several data path widths are present within the control section.

MEMORIES

Name	Use	Cell size	Number of cells
Main memory	Holds the PDP-11 level 2 programs and data; this is the only memory the level 2 programmer is aware of	8	16K-248K
CPU local storage	Holds the level 2 machine's 8 general registers and 8 temporary variables used by the microprogram	16	16
Micromemory	Holds the microprogram	56	256

REGISTERS

Name	Width	Use
B	16	Temporary storage to set up left ALU input in advance
D	16	ALU output is stored here
PS	8	Program status bits, including the 4 condition code bits NZVC which are set if the previous result was Negative, was Zero, Overflowed, or had a Carry out of bit 15 (leftmost bit)
BA	18	UNIBUS addresses are kept here before being transferred onto the UNIBUS; this is a generalized MAR, addressing both memory and i/o device registers. (BA stands for Bus Address)
IR	16	Holds the current level 2 instruction being interpreted
MPC	8	Microprogram counter; this is also the MAR for the micromemory
MIR	48	Microinstruction register; this is also the MBR for the micromemory; microinstructions are kept here while being executed

MULTIPLEXORS

Name	Number of inputs	Use
B MUX	2	Selects the ALU left input from either the D register, or 1 of the wired in constants (1, 2, various trap addresses, etc.)
D MUX	4	Selects the input to the bus feeding the D register, PS register, local storage, and IR
BA MUX	2	Selects the input to the BA register from either the ALU output or the local storage
BUT MUX	2	Selects bits to input to the microprogram counter

Note: Multiplexors are essentially switches. They have no storage capacity.

FUNCTIONAL UNITS

Name	Output width	Use
ALU	16	Performs any 1 of 16 arithmetic or 16 boolean functions
Shift unit	16	Performs 1-bit right shifts
Branch control	6	Contains the circuits used to determine the address of the next microinstruction to be executed

Fig. 4-30. Summary of the components of the PDP-11/40's microprogramming level.

The CPU has two internal memories in addition to the main memory, which is external. The CPU local storage holds 16 words of 16 bits each. Words 0 to 7 are the level 2 general registers. Words 8 to 15 are temporary variables used by the microprogram. CPU local storage words can be gated into the right ALU input, the B register, or the bus address register. The CPU local storage is loaded from the ALU output and from the UNIBUS. The microprogram is kept in a 256-word micromemory.

Four multiplexers are shown in Fig. 4-29. They are indicated by the label MUX. These multiplexers all have two or more inputs and one output. They are, in fact, simply switches that select one of the inputs to be used as output. Bits in each microinstruction determine which of the inputs is to be output. Unlike registers, the multiplexers do not have any storage capacity and remember neither the last data word held nor the last input line selected.

The ALU can perform all 16 boolean functions of two variables. It can also perform 16 arithmetic operations on its inputs, A and B, with the carry bit or the constant +1 added or not added to the result under microinstruction control. Some of the more important functions are

A + B, A − B, A − 1, A + 1, A + B + 1, A − B − 1, A + A, A + A + 1, −1, 0

Most of the remaining functions involve mixtures of boolean and arithmetic operations, such as A + (A AND NOT B).

The shift unit can perform only one-bit right shifts, with the carry bit inserted into the leftmost bit. However, left shifts of one bit can be achieved by using the ALU function A + A.

4.12.2. UNIBUS Operation

The UNIBUS plays an essential role in the communication between the CPU and the rest of the PDP-11/40 system. The CPU uses the UNIBUS to read and write main memory and to access all the i/o device status and buffer registers directly. Because every i/o device register has a unique address, the CPU can address any one of them the same way it addresses memory.

For example, when the microprogram puts the number 4000 in the bus address register and initiates a bus READ, the contents of the memory word at address 4000 can be read into the CPU via the D multiplexer. However, if the number 777550 (octal) is put into the bus address register and a bus READ initiated, the contents of the paper-tape-reader status register [see Fig. 3-35(a)] is available at the D multiplexer. All bus addresses from 760000 to 777777 (octal) correspond to the i/o and other registers, and not to memory addresses. Thus the maximum addressable memory is limited to 760000 (octal) bytes.

The UNIBUS consists of 56 parallel wires. The fact that the microinstructions are also 56 bits is purely coincidental. The 56 UNIBUS lines are divided into groups as shown in the table.

Size	Function
18	specifies a unique UNIBUS address
16	data coming from or going to a UNIBUS device
2	parity bits, one for each byte of data
1	master synchronization
1	slave synchronization
2	type of operation (read word, write word, write byte, or modify)
5	bus request (from a slave to the master)
5	bus request granted (from the master to a slave)
1	selection acknowledge (slave accepting bus grant)
1	begin the CPU interrupt sequence
1	bus busy
1	initialize all devices and clear all error flags
2	ac or dc voltage too low (begin power failure interrupt)

Whenever two system components are communicating via the UNIBUS, one of them is active and issues instructions (the master) and the other is passive and obeys the instructions (the slave). A typical example is the CPU as master directing the main memory as slave. When the microprogram wants to read or write from memory or an i/o register, it puts the desired address on the UNIBUS address lines and sets the master synchronization line to 1 to indicate that it wants something.

As the signal propagates down the UNIBUS, each device (including main memory) checks to see if the specified address belongs to it. If so, it acknowledges this fact by setting a 1 on the slave synchronization line. When the bus master detects the slave synchronization bit, it changes the master synchronization to 0, and the transfer begins. Because the master waits for an explicit acknowledgement signal from the slave, it is possible to have a collection of devices on the UNIBUS with different speeds.

The bus request lines are used by i/o devices to request permission to become bus master (e.g., a disk wanting to transfer a word to main memory). Each line is for a different priority. The bus grant lines allow the CPU to grant or deny such requests, depending on its own priority.

In order to initiate a UNIBUS transfer, the microprogram first puts an address in the BA register. Then it gates this address onto the UNIBUS address lines, while at the same time specifying the operation desired and signaling the UNIBUS to start. If a READ has been requested, the word read is sent to the D multiplexer, from which it can be sent to the B register, CPU local storage, or instruction register. If a WRITE WORD or WRITE BYTE operation has been requested, all or half of the D register is written. The MODIFY operation is a high-speed way of allowing the microprogram to read a memory word, modify it, and write it back. The level 2 instruction INC (increment a memory word by 1), among others, uses this UNIBUS command.

Although PDP-11 addresses are only 16 bits, UNIBUS addresses are 18 bits. The extra two bits are useful in time-sharing systems where there is more than one user

in main memory. Addressing limitations restrict each user program to no more than 2^{16} bytes, but all the user programs together may easily exceed this amount. The optional virtual memory hardware provides a way to map each user's address space onto a different part of the UNIBUS address space. Addresses between 160000 and 177777 (octal) can be supplied with two high-order ones by the hardware to allow programs to access the i/o registers.

4.12.3. PDP-11/40 Microinstructions

All microinstructions have exactly the same format, as shown in Fig. 4-31. This situation is in sharp contrast to the IBM 3125 microprocessor level, which has 13 different microinstructions, each with several variations. Moreover, the PDP-11/40 microinstructions are nearly three times as long as those of the 3125. The PDP-11/40 microinstructions are predominantly horizontal, with 26 distinct fields, some of which control individual gates.

The 26 fields in the microinstructions can be further classified into seven groups.

Group 1 (CLKL, CLKOFF) controls the clock. Several subcycle lengths can be selected, and the clock can be turned off. A long clock time is needed when a UNIBUS operation is initiated, because the CPU has no way of telling what kind of device it is addressing, much less how long it will take to get a reply.

Group 2 (CLKIR, WRH, WRL, CLKB, CLKD, SPS) controls the gating of data into registers. The ability to specify the upper and lower bytes of the CPU local storage input is needed to handle the level 2 machine's extensive byte manipulation instructions.

Group 3 (CLKBA, CBUS, BGBUS) controls operation of the UNIBUS, including the specification of the type of transfer to be performed. When a byte is to be written to an even address, the low-order byte of the D register is used. When a byte is to be written to an odd address, the high-order byte of the D register is used.

Group 4 (DAD, SALUM, SALU) determines which of the ALU functions is to be performed. The SALUM bit tells whether boolean or arithmetic mode is to be used, and SALU specifies the function. However, DAD can modify the operation, for example, by causing the ALU function to be taken from the instruction register instead of the SALU field or by forcing a carry bit to be added to the ALU result.

Group 5 (SBC, SBMH, SBML, SDM, SBAM) controls the three arithmetic section multiplexers. The SBMH and SBML fields are each responsible for one byte of the B register output. Thus either byte can serve as either upper or lower input; moreover, the bytes can also be swapped. If the B constants input is selected, the SBC field determines which of the constants it will be. The constants 1 and 2 are used primarily for autoincrement and autodecrement addressing, and for incrementing the program counter.

Group 6 (SRS, SRD, SRBA, SRI, RIF) is used when the CPU local storage is being read from or written into. If the SRS bit is a 1, one of the eight level 2 general registers is selected according to the number in bits 6 to 8 of the instruction register. These are the source register bits of the double-operand level 2 instructions [see

Field name	Bits	Function
CLKL	2	Clock length. Allows the microprogram to control subcycle length.
CLKOFF	1	Clock off. Allows the microprogram to turn the clock off.
CLKIR	1	Gate a word from the UNIBUS into IR for examination and decoding.
WRH	1	Gate the high order byte of the D MUX into CPU local storage.
WRL	1	Gate the low order byte of the D MUX into CPU local storage.
CLKB	1	Gate the D MUX output into the B register.
CLKD	1	Gate the ALU output into the D register.
CLKBA	1	Gate the BA register onto the UNIBUS address lines.
CBUS	2	Specify the type of UNIBUS transfer: read a word; write a word; write a byte; and read a word modify it and write it back.
BGBUS	1	Start a UNIBUS operation.
DAD	4	Used to alter the ALU function depending upon the level 2 instruction being executed.
SPS	3	Control loading of bits into the PS register.
SALUM	1	Selects ALU mode: arithmetic or boolean.
SALU	4	Selects the ALU function. SALU, SALUM, DAD, and the level 2 instruction opcode together determine the ALU function.
SBC	4	Selects a constant to be gated into the ALU left input via the B MUX.
SBMH	2	Controls the B switch for the high order byte: true B, sign extension of low order byte, low order byte of B, or high byte of a constant.
SBML	2	Controls the low order byte passing through the B switch: either byte of B, or the low order byte of one of the constants.
SDM	2	Selects 1 of the D MUX inputs: UNIBUS data, ALU output, ALU output shifted right 1 bit (with carry bit in bit 15) or CPU local storage.
SBAM	1	Selects input to the BA register: ALU output or CPU local storage.
UBF	5	Selects condition to be tested. This field controls branching within the microprogram.
SRS	1	Use source register specified in level 2 instruction as CPU local storage register addressed.
SRD	1	Use destination register specified in level 2 instruction as CPU local storage register addressed.
SRBA	1	Use the low order 4 bits of the BA register to select CPU local storage.
SRI	1	Use the RIF field of the microinstruction to address CPU local storage.
RIF	4	Specifies one of the 16 CPU local storage registers.
UPF	8	Address of the next microinstruction.

Fig. 4-31. The fields in the PDP-11/40 microinstructions.

Fig. 3-30a)]. The SRD bit is similar, but it uses bits 0 to 2 instead of 6 to 8. The SRI bit enables an immediate four-bit address contained in the RIF field to be used. Its purpose is to access the scratch registers.

Group 7 (UBF, UPF) controls the sequence of microinstructions executed. Unlike the IBM 3125, the PDP-11/40 microinstructions located in consecutive micromemory words are not executed sequentially. Each microinstruction contains an eight-bit field (UPF) specifying one of the 256 words in the micromemory. If UBF = 0, UPF is the address of the next microinstruction.

However, if UBF is not 0, one of 31 different branch conditions is enabled, and the UPF address can be modified. Each of the 31 nonzero UBF branch conditions results in six bits being computed as functions of the bits in the instruction register, the status of the console switches, the presence or absence of bus requests, and other miscellaneous bits.

These six bits are OR'ed with the low-order six bits of UPF so as to produce the address to branch to. Most of the conditions are very complicated functions of the IR bits. Note that because the six computed bits are OR'ed with the original bits, only forward branches are possible. Because all six bits are determined independently, each microinstruction is potentially a 64-way branch. The result is highly efficient for interpreting PDP-11 instructions because the tests (which are hard wired) check for precisely those bit combinations that are relevant to the PDP-11 interpreter and no others.

As an example of one of the simplest tests, when UBF = 17 (octal), then bit 1 of the branch address is OR'ed with bits 3, 9, 10, and 11 of IR. Because these tests are so specific to the level 2 instruction bit patterns, it would be difficult to use the host processor to interpret any machine other than a PDP-11.

PDP-11/40 branch microinstructions (UBF > 0) branch in a somewhat peculiar way. If a branch microinstruction at address A specifies the microinstruction at B as its successor, B is executed whether or not the branch occurs. The execution of B occurs because the CPU is overlapped, and the fetching of B has already begun by the time it is known that a branch will take place. Instead of trying hastily to abort the fetching of B in the middle, the PDP-11/40 designers decided to let it occur and to let B be executed regardless of the outcome of A's test. The six-bit address modification logic does not use the UPF field contained in A as the base address but uses the UPF field contained in B instead. The actual branch takes effect after B has been executed. If every branch microinstruction were followed by a NO OPERATION, the microprogram would be simplified, but the interpreter is slowed down. The 3125 has the same property: procedure returns take place one microinstruction after being specified.

It is noteworthy that the entire PDP-11 interpreter fits into only 256 words of micromemory. The highly PDP-11 oriented microprogramming level architecture makes such a compact microprogram possible. The PDP-11/40 designers clearly never intended that the CPU be microprogrammed to interpret any machine differing appreciably from a PDP-11.

In contrast, the 370 interpreter on the 3125 is far larger, partly because of the

size of the 370 level 2 instruction set and partly because of the generality of the 3125 architecture. The 3125 was designed to interpret not only 370 level 2 programs but also IBM 1401, IBM 1440, IBM 1460, and IBM 360/20 programs, for the benefit of customers who traded in these computers for a 370/125.

4.13. THE BURROUGHS B1700

The Burroughs B1700 series represents a radically different approach to computer organization than the 370 or PDP-11. On the latter two machines, first the level 2 virtual machines were designed and then the level 1 architectures were chosen to interpret them efficiently. The PDP-11/40 microprogramming level was designed in such a way as to be excellent at interpreting the PDP-11 conventional machine level, but to be nearly useless for interpreting anything else. The 3125 is more general, but it was optimized toward 370-like or 1401-like computers.

The B1700 series, on the other hand, is a family of computers with a very general microprogramming level. They can efficiently interpret PDP-11s, 370s, and a large variety of other virtual machines, spanning a far wider class of architectures than these two. The B1700 microprogramming level has no preference as to the word length, instruction format or instruction set of the level 2 machines they can support. It can handle a 23-bit target machine as easily as a 16-bit target machine.

This enormous generality has two major consequences:

1. The designers of level 2 machines are free to choose whatever instructions and lengths they really need.
2. It is not only possible but also expected that widely different applications will use widely different level 2 machines.

For example, on the 370, both the FORTRAN and COBOL compilers produce code for the same (level 3) target machine, even though FORTRAN and COBOL are very different languages. The 370 level 2 and level 3 designs were chosen as a compromise among the conflicting requirements of FORTRAN, COBOL, the operating system, and many other languages and programs.

The B1700, on the other hand, has a FORTRAN compiler that generates code for a target machine specifically designed for FORTRAN, and a COBOL compiler that generates code for a total different target machine specifically designed for COBOL. Each of these virtual machines has a different interpreter, of course. A time-sharing system on the B1700 could easily allow each user to have a different level 2 (and level 3) machine. Switching machines could be accomplished by having several interpreters in memory and by having control simply jump from interpreter to interpreter, depending on the level 2 machine currently in use.

Tailoring the level 2 (and level 3) machines to the application, rather than contorting the problem to fit the machines, has the obvious advantages mentioned earlier, but it also has certain drawbacks. The most serious problem is that the

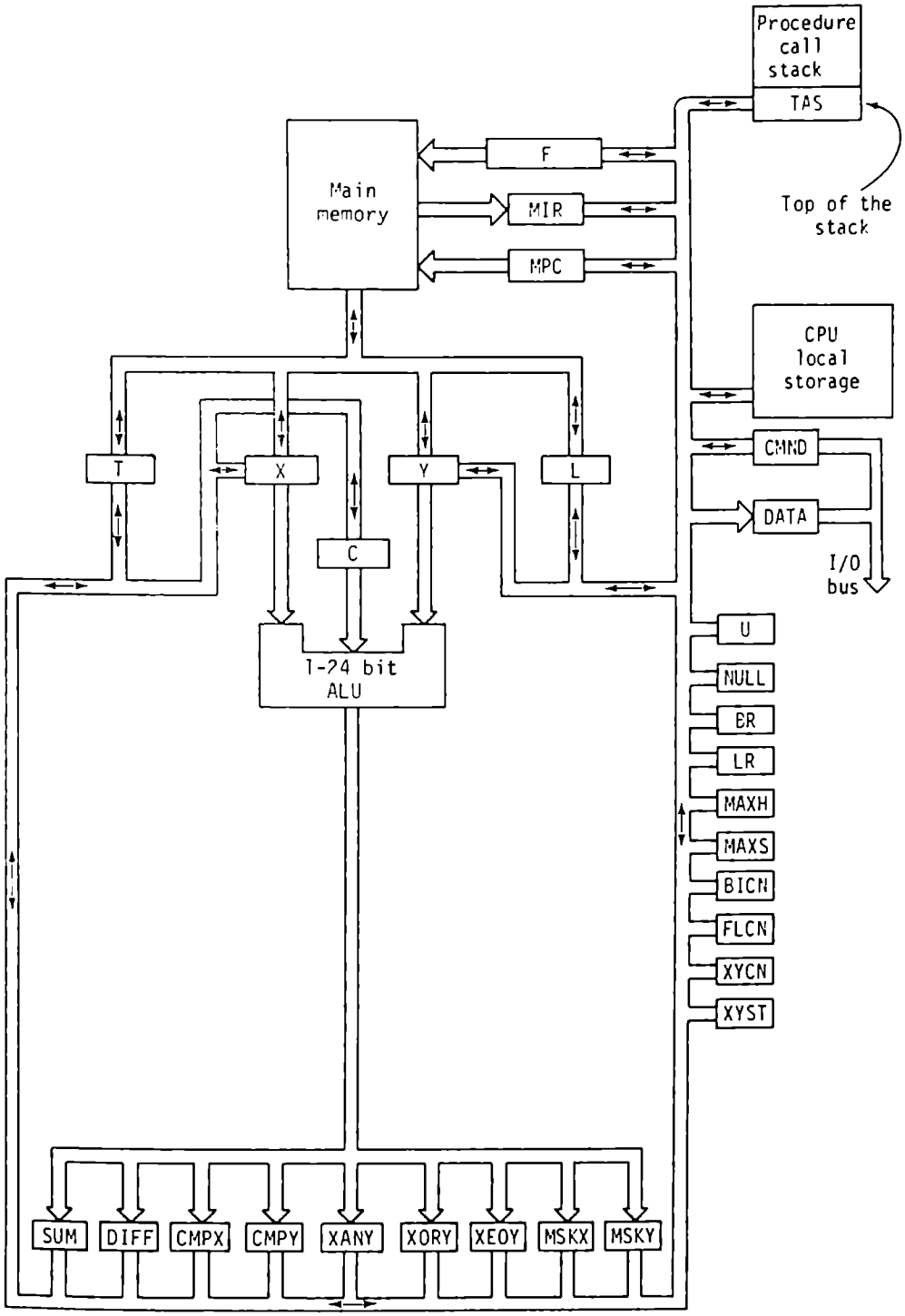


Fig. 4-32. A simplified diagram of the B1700 registers and data paths.

microprogramming level architecture must be flexible enough to be able to efficiently interpret virtual machines that may be needed in the future but that have not even been thought of yet. In the following sections it will be shown how Burroughs has dealt with this problem.

Like the 370 and PDP-11, the B1700 is a family of computers. However, unlike the 370's and PDP-11's, each of whose models has a totally unique microprogramming level architecture, the differences between members of the B1700 family are very minor. The larger machines have a few extra microinstructions, registers, and features to optimize the handling of certain special situations. The discussion following applies specifically to the B1712, one of the simpler models.

4.13.1. Architecture of the B1700

Figure 4-32 is a simplified diagram of the components and data paths at the B1700 microprogramming level. There is one bus that connects every register to every other register so as to allow arbitrary register-to-register transfers. The ALU input, ALU output, and memory buffer registers are all 24 bits wide. However, under program control they can be made to function as though they were smaller. Four of the registers shown are divided into several subregisters, as illustrated in Fig. 4-33. These subregisters can be addressed in the same way as the full registers. For example, the MOVE REGISTER microinstruction can move L to T, or it can move

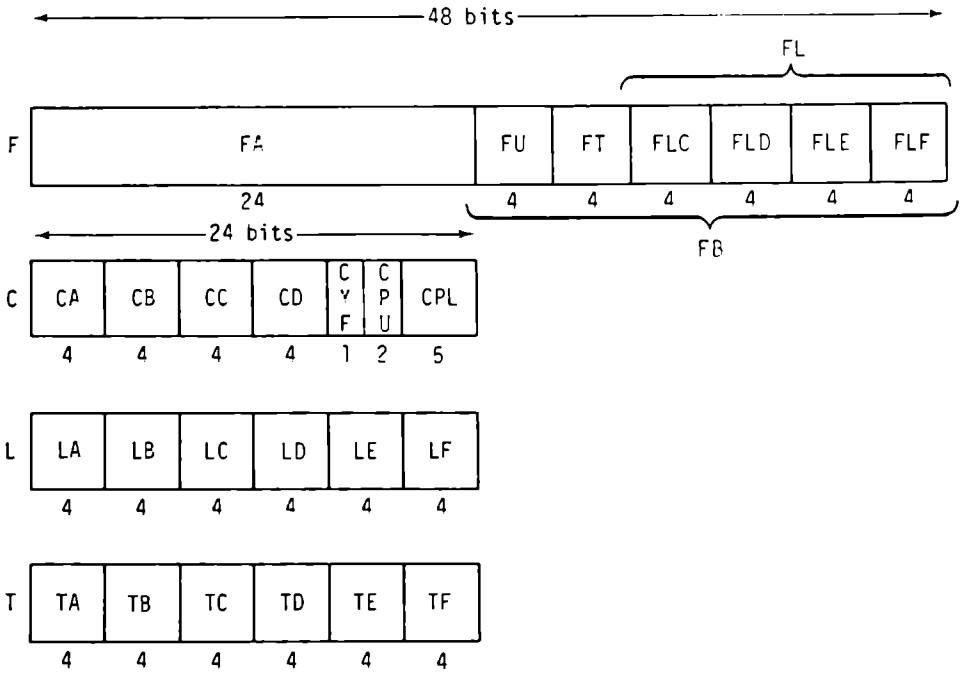


Fig. 4-33. Four of the registers of Fig. 4-32 are divided into subregisters.

LC to TF, among many other possibilities. There is a separate ALU (not shown in Fig. 4-32) for manipulating four-bit registers.

The BICN, FLCN, XYCN, and XYST (pseudo) registers contain continuous readouts of certain conditions within the B1700. For example, bit 0 of XYCN is set to 1 by the hardware whenever $X > Y$ is true; bit 1 of XYCN is set to 1 whenever $X < Y$ is true; bit 2 of XYCN is set to 1 whenever $X = Y$ is true. Whenever X or Y is changed, the bits in XYCN are changed automatically. By testing bits in XYCN, the microprogram can determine the arithmetic relationship between X and Y efficiently.

The model 1712 system does not have a separate micromemory for microprograms, although some of the larger models do have a special high-speed micromemory. B1712 microprograms reside in the main memory, along with the target machine programs.

The B1700 ALU works differently than the IBM 3125 or the PDP-11/40 ALUs. The ALU can compute any one of nine different functions of its inputs, X and Y. The ALU has nine output (pseudo) registers, as shown in Fig. 4-32. On every microinstruction, all nine functions are computed and the results stored in the corresponding registers. Consequently, none of the microinstructions contains any "ALU function" bits, as is the case on the 3125 and PDP-11/40. To store the sum of X and Y in register L, a microinstruction simply moves the SUM register to L. The functions of all the ALU output registers are given in Fig. 4-34.

The B1700 has the ability to vary its ALU width under microprogram control. The CPL field of the C register contains a number between 1 and 24 that causes the ALU to function as though it had that many bits, instead of 24. For example, if CPL 16, the ALU functions as a 16-bit-wide ALU. High-order input bits are ignored, and high-order output bits are set to 0. Consider the PL/I declaration

```
DECLARE WEIRDO FIXED BINARY (19,0);
```

A PL/I compiler for the B1700 could output information to the run-time interpreter, telling it that 19-bit integer arithmetic is going to be needed. The interpreter could then load the constant 19 into CPL and have the equivalent of a 19-bit ALU. The "word length" of the target machine can be set dynamically before a program begins executing, or it can be changed during its execution. ALU widths greater than 24 bits can be simulated by using multiple-precision arithmetic. A field in the C register selects either binary or decimal arithmetic. These features provide the B1700 with the capability of supporting a wide class of target machines.

In order to be able to interpret a wide variety of instruction formats, a microprogram needs a method of fetching (and storing) arbitrary-length bit strings from main memory. To do so, the B1700 has a one-bit cell size. In other words, every single bit in the main memory has its own unique address. The B1700 has a microinstruction that can load X, Y, T, or L with any sequence of 1 to 24 consecutive bits, beginning at any specified bit in main memory. Fetching an arbitrary bit string on the 3125 would require first fetching an integral number of bytes, then shifting the bytes to right-adjust the bit string in a register, and, finally, masking out all the unwanted bits.

MEMORIES

Name	Use	Cell size	Number of cells
Main memory	Holds level 2 programs and data, and also the microprogram(s)	1	128K-2048K
CPU local storage	Microprogram variables	48	16
Stack	Return addresses for micro procedure calls	24	16

REGISTERS AND SUBREGISTERS

Name	Width	Use
X	24	ALU input and memory buffer register
Y	24	ALU input and memory buffer register
T	24	Bit fields in T can be easily extracted; also memory buffer register
L	24	General working storage and memory buffer register
C	24	CA and CB are general purpose; the other fields have specific functions
CC	4	I/O devices set bits here to signal completion
CD	4	Records memory parity errors and protection violations
CYF	1	ALU carry bit
CPU	2	Selects decimal or 2's complement binary arithmetic
CPL	5	Determines ALU width from 1 to 24 bits
SUM	24	ALU output = X + Y (binary or decimal depending on CPU)
DIFF	24	ALU output = X - Y (binary or decimal depending on CPU)
CMPX	24	ALU output = 1's complement of X (NOT X)
CMPY	24	ALU output = 1's complement of Y (NOT Y)
XANY	24	ALU output = X AND Y
XORY	24	ALU output = X OR Y
XEOY	24	ALU output = X EXCLUSIVE OR Y
MSKX	24	Low order N bits of X, high order bits set to 0 (N is in CPL)
MSKY	24	Low order N bits of Y, high order bits set to 0 (N is in CPL)
F	48	Generalized memory address register
FA	24	Main memory address (addressable to the bit)
FL	16	Total field length of the current field
MIR	16	Microinstruction register (holds current microinstruction)
MPC	19	Micro program counter
NULL	24	24 0-bits used as a source of zeroes
CMND	24	Commands to the I/O equipment are put here
DATA	24	Communication of data to and from the I/O equipment
BICN	4	Carry and borrow bits, and least significant unit of Y
FLCN	4	Bits telling if FL = SFL, FL > SFL, FL < SFL, FL ≠ 0 (SFL is in local storage)
XYCN	4	Most significant bit of X, and the 3 values X = Y, X < Y, X > Y
XYST	4	Least significant unit of X, interrupt pending, Y ≠ 0, X ≠ 0
MAXM	24	Amount of micro memory installed (always 0 for model 1712)
MAXS	24	Amount of main memory installed
U	16	Used for loading microprograms from the tape cassette
BR	24	Main memory base register (used with LR for protection)
LR	24	Main memory limit register (used with BR for protection)

Fig. 4-34. The registers and some of the subregisters used on the B1700.

The CPU local storage on the B1700 consists of 16 48-bit words. Each 24-bit half word has its own address, however, so data can be moved into or out of it in units of 48 or 24 bits. Moves to or from T, X, Y, C, L, or the ALU outputs use a half word, while the complete F register requires a full 48 bits.

4.13.2. The B1700 Instruction Set

A brief summary of all the B1712 microinstructions is given in Fig. 4-35. A few of the instructions have minor variations or secondary effects that are not listed. Except for number 4, all microinstructions are 16 bits. An expanding opcode scheme is used to accomodate 4, 8, 12, and 16-bit opcodes. The B1712 has a total of 55 addressable registers and subregisters, 53 of which are listed in Figs. 4-33 or 4-34. (F and CYF are not addressable as source or destination registers.) The other two are console registers.

The REGISTER MOVE microinstructions can, with a few exceptions, move any register to any other register. If the source is longer than the destination, only the low-order bits are moved. If the source is shorter than the destination, the source is loaded right-adjusted with the high-order bits set to 0. Six bits are used to specify the source, and six are used to specify the destination. Some of the unused bit combinations correspond to registers on the other models of the B1700 series.

There are several ways to specify a transfer to or from main memory. A bit in microinstruction 6 tells whether FA specifies the first or last bit in the field to be read. The field length can be taken either from the microinstruction itself or from CPL. Furthermore, FA or FL can be incremented or decremented. Both shifting and rotating any amount up to 48 bits are possible by using X and Y as a combined 48-bit register. The EXTRACT instruction allows arbitrary fields within T to be removed and copied into another register right-adjusted without affecting the contents of T.

As an example of how extracting is used, consider the interpretation of a 16-bit instruction containing several short fields, such as Fig. 3-30(a). Assume that T already contains the entire 16-bit instruction. First, the four-bit opcode could be extracted and loaded into X, right-adjusted, and filled out with high-order zeros. Then the address of a 16-entry table containing microbranches to the interpreters for each opcode could be loaded into Y. The third microinstruction could move SUM to MPC, effecting a 16-way branch for efficient instruction decoding. Furthermore, the original instruction is still available in T.

Note that such a 2^N-way branch also only requires three microinstructions for opcodes of 8 bits, or 5 bits, or 17 bits, or any other number of bits up to 24. In contrast, on many other computers, decoding an arbitrary-length opcode would first require copying the instruction to preserve the original, then right shifting the copy, and, finally, masking off any unwanted bits. Because an interpreter executes the instruction decode sequence so often, even a small saving here can dramatically affect the execution speed of the target machine.

Another way to perform instruction decoding also exists. First, the opcode is

Data movement microinstructions

1. Move a register to another register.
2. Move a CPU local storage 24-bit halfword to or from a register.
3. Move an 8-bit immediate field to a register.
4. Move a 24-bit immediate field to a register (32-bit microinstruction).
5. Store F in, and reload F from, the CPU local storage (including swap F).
6. Transfer 1-24 bits between main memory and X, Y, L, or T as specified by FA.

Shift/rotate microinstructions

7. Shift or rotate T 1-24 bits left and move the result to another register.
8. Extract a field from T and move it right adjusted with 0 fill to another register.
9. Shift or rotate either X or Y by 1-24 bits left or right.
10. Treat XY as a 48-bit register and shift or rotate it left or right by 1-48 bits.
11. Normalize X by left shifting.

Skip/branch/call microinstructions

12. Test any bit in a 4-bit register and skip up to 16 microinstructions if it is 0.
13. Test any bit in a 4-bit register and skip up to 16 microinstructions if it is 1.
14. Test selected bits in a 4-bit register for several conditions and skip.
15. Unconditional branch forward up to 4095 microinstructions.
16. Unconditional branch backwards up to 4095 microinstructions.
17. Procedure cell. Branch forward up to 4K and save return address on the stack.
18. Procedure call. Branch backward up to 4K and save return address on the stack.

Other microinstructions

19. Perform operation (LOAD, +, -, AND, OR, EX OR) on 4-bit register, using an immediate field.
20. Increment or decrement FA or FL or both by a 4-bit immediate field.
21. Add or subtract a CPU local storage 24-bit half word to (or from) FA.
22. Set CYF to 0, 1, the carry from X + Y or the borrow from X - Y.
23. Set subregisters of C conditionally.
24. Start or stop the tape cassette used to load microprograms.
25. No operation.
26. Halt.
27. Monitor (sets voltage levels for use by the maintenance personnel).

Fig. 4-35. Brief descriptions of all the B1712 microinstructions.

isolated in X, as above, using the EXTRACT microinstruction. The microinstruction following the extract moves X to MIR. However, due to overlap on the B1700, the microinstruction following the MOVE X TO MIR will be fetched concurrently with the execution of the move itself. This means that the MIR has two sources: X and the next microinstruction. These two sources are OR'ed together. If the next microinstruction is a BRANCH FORWARD by an amount 0, the resulting microinstruc-

tion assembled in MIR will be a BRANCH FORWARD by the amount in X. A table of branches to the proper addresses follows the BRANCH FORWARD 0. The decoding mechanism can be described as follows:

```
EXTRACT 4 BITS FROM T(O) TO X  /* PICK OFF LEFTMOST 4 BITS */
MOVE X TO MIR                  /* LOAD MICROINSTRUCTION REGISTER */
BRANCH FORWARD O               /* TO BE OR'ED WITH X IN MIR */
GO TO OPCODEO                  /* 16 ENTRY BRANCH TABLE */
GO TO OPCODE1
.
.
.
GO TO OPCODE15
```

All the skips and branches, as well as the procedure calls, use self-relative addressing. The address to branch to is specified by giving its distance from the branching microinstruction. Among other properties, this means that an interpreter can be easily moved without invalidating it. If direct addressing had been employed, moving an interpreter would have been much more difficult. Since the ability to switch target machines rapidly depends on the ability to switch interpreters rapidly, being able to swap an interpreter to secondary memory and then reload it at a different address is useful.

PROBLEMS

1. Give a short explanation of the use of the following:
   a. registers
   b. buses
   c. gates
   d. clocks
2. Draw a register diagram showing the transfers involved in performing a left rotate of one bit. Show each step separately.
3. Draw gate sequences analogous to Fig. 4-12 for the PUSH instruction.
4. Draw gate sequences analogous to Fig. 4-12 for the POP instruction.
5. What is the result of opening gates 6, 15, and 36 during one cycle of the Fig. 4-10 CPU?
6. If the host level of Fig. 4-10 did not have a register with the constant -1, how could the stack pointer be decremented?
7. The target level of Figs. 4-8 and 4-9 is to be modified to have indirect addressing (planned obsolescence). Bit 12 is to be used as the indirect bit, and bits 0 to 11 provide an address for the new 4K memory. What changes should be made to the host machine so as to ease the task of interpreting the new level?
8. Add a new instruction to the target level of Fig. 4-9 to pop two operands off the stack, compare them, and skip the next instruction if they are equal.
9. The manufacturer of the computer of Fig. 4-10 has just received a telegram from its most important customer. The telegram reads: "Must have target machine instruction

to shift top of stack right 1 bit with sign extension. Needed tomorrow morning 8 am. Please implement and send new microprogram by telegram." Your job is to implement the right shift.

10. Add a new instruction to the example target machine that replaces the top of the stack by its 2's complement.

11. Add an EXCLUSIVE OR instruction to the repertoire of Fig. 4-9. This instruction pops the top two words off the stack as operands and pushes the result back on the stack. Use opcode 160005 (octal). (*Hint:* Build up the result one bit at a time, right to left. Put the operands in the A and C registers. Add A and C and then send the result to MBR. The low-order bit of MBR is the low-order bit of the result. Insert this into the high-order bit of D; then right shift A, C, and D one bit and repeat.)

12. Modify the microprogram of Fig. 4-15 to detect multiplication overflow. If the result of a multiplication is less than −32,768 or greater than 32,767, the target machine should trap to location 1.

13. To speed up looping, add a new 0-address instruction, LOOP, to the target machine of Fig. 4-9. This instruction increments memory location 0 by 1 and tests the result. If the result is greater than or equal to 0, the next target machine instruction is skipped.

14. What changes should be made to the example host machine so as to make detecting addition overflow easy? Describe, in words, how the information that an overflow had occurred could be communicated to a target machine program.

15. If the microprogram counter of Fig. 4-10 were accessible as an adder input and output, would the target-machine-instruction decoding logic be speeded up? What other changes would be needed? Discuss.

16. If the example target machine were 1's complement instead of 2's complement, and the ALU were also 1's complement, would the microprogram need to be changed? If so, in what ways? Assume that the ALU does not convert −0 to +0.

17. Devise a better division algorithm than the one given in the text. (*Hint*: Use nonrestoring division. That is, when the dividend becomes negative, instead of restoring it to its original value, perform the shift and add instead of subtracting on the next cycle.) How does the sign of the dividend after the addition relate to the next quotient bit?

18. Write a program for the target machine of Fig. 4-9 to compute the first one hundred prime numbers.

19. Do you think the CPU of Fig. 4-10 would be a good choice for a target machine of a PDP-11 or other machine with many short fields in its instructions? Explain.

20. What is the execution time of each of the target machine instructions of Fig. 4-9 in units of the microinstruction time. If the target machine instruction execution time is variable, on what does it depend?

21. Design the microprogramming level to be used for interpreting the following target machine (a simplified PDP-8). The word length is 12 bits. All instructions are also 12 bits, with the opcode in bits 9 to 11. Opcodes 0 to 5 have a 3-bit opcode field, 1 indirect bit, and an 8-bit address field. Indirect addressing allows programs to access up to 4K words. Arithmetic is performed on a 12-bit accumulator.

    The instructions with no addresses share opcode 7. If bit 8 is a 0, then bits 0 to 7 can specify up to eight instructions to be performed, more than one of which is allowed

in an instruction. Each of the eight instructions is enabled by a different bit between 0 and 7. If bit 8 of an instruction with opcode 7 is a 1, each of the low-order bits selects a condition to be tested. If any of the conditions is true, the next instruction is skipped. The instructions are as follows (M is a memory word, and AC is the accumulator):

Mnemonic	Opcode	Meaning
AND	0	AC AC AND M
TAD	1	AC AC + M (2's complement add)
ISZ	2	M M + 1; if M 0 after incrementing, skip the next instruction
DCA	3	M AC; AC 0
JMS	4	M = return address, then jump to M + 1
JMP	5	jump
IOT	6	I/O. 3-bit OP field specifies operation, and 6-bit field selects a device. Data is transmitted to or from the AC

Group 0 of opcode 7

IAC	AC AC + 1
CLA	AC 0
CMA	AC 1's complement of AC
BSW	the high-and low-order 6 bits are swapped
RAL	rotate AC 1 bit left
RTL	rotate AC 2 bits left
RAR	rotate AC 1 bit right
RTR	rotate AC 2 bits right

Group 1 of opcode 7

SMA	skip minus AC
SPA	skip plus AC
SZA	skip zero AC
SNA	skip nonzero AC

22. Write the microprogram for the target machine of Problem 21.

23. Design a CPU architecture to support a target machine that will evaluate boolean expressions, such as ((P OR Q) AND (P OR R NAND NOT S) NOR P) OR Q. The target machine has a stack and hardware instructions to pop two operands off the stack, perform a boolean operation, and push the result back on the stack. The target machine is to have instructions for AND, OR, EXCLUSIVE OR, NAND, EQUIVALENCE, and NOT (monadic). The ALU has only one function, NAND. Specify the details of the target level and microprogramming level in a form similar to the example in this chapter.

24. Write the microprogram for Problem 23.

25. A certain micromemory contains 1024 words of 108 bits each. If only 128 different bit combinations are used, how many bits can be saved, using a nanomemory?

26. Devise a modification to the architecture of Fig. 4-22 to allow one ALU operation and one register-to-register transfer to occur during the same cycle.

27. The president of the Nano-micro-milli Memory Corporation has been so impressed by the sales of their nanomemories that he has considered adding a picomemory to the product line. The picomemory would be referenced by the nanomemory, the same way the nanomemory is referenced by the micromemory; that is, the nanomemory would contain picoaddresses, just as the micromemory contains nanoaddresses. You are the vice president in charge of getting rid of dumb ideas. What do you think of this proposal?

28. If the microprogram of Fig. 4-15 were to be rewritten for the machine of Fig. 4-22, what changes would need to be made?

29. If you were to feed Fig. 4-15 to a PL/I compiler (or with appropriate changes, to some other compiler) that produced code for some level 2 machine, such as that of the 370, PDP-11, or Cyber 70, do you think the number of bits in the generated program would be more or less than for the example host machine? Discuss the kinds of conventional machine level instructions it might generate for each type of microinstruction.

30. Consider a single chip CPU using a vertical organization of the type shown in Fig. 4-22, only with 64 registers instead of 16. The microinstruction decoding logic for the machine is on a different chip. How many wires must the CPU chip have; that is, how many bits does it take to specify the actions for one cycle? Now consider a horizontal-type microprocessor with 64 registers, in which each register has but four inputs and four outputs. How many wires would this chip need? Do you expect most single chip CPUs to be vertical or horizontal? Does your conclusion change if the micromemory and decoding logic are located on the same chip as the CPU?

31. The IBM 3145 CPU has 32-bit data paths, compared to 16-bit data paths for the 3125. How does this fact help it achieve a higher speed for interpreting 370 conventional machine language programs?

32. The multiplication time of the IBM 370/125 depends on the number of 1 bits in the multiplier. Give a plausible explanation.

5 THE OPERATING SYSTEM MACHINE LEVEL

In the previous chapter it was shown how an interpreter running at the microprogramming level (level 1) could execute programs written for the conventional machine level (level 2). On a microprogrammed computer, the conventional machine level instructions, such as procedure call, multiplication, and loop, are not carried out directly by the hardware but rather are fetched, examined, and executed as a series of small steps by the microprogram. The level 2 machine can be programmed by people who know nothing at all about the level 1 machine and its interpreter. As far as they are concerned, the level 2 machine can be used as though it were the real hardware.

Just as an interpreter running on the level 1 machine can interpret programs written in level 2 machine language, an interpreter running on the level 2 machine can interpret programs written in level 3 machine language. For historical reasons (see Section 1.3), the interpreter running on the level 2 machine that supports the level 3 machine is called an operating system, as shown in Fig. 5-1. Therefore we will call level 3 the "operating system machine level," for lack of a generally accepted term.

5.1. IMPLEMENTATION OF THE OPERATING SYSTEM MACHINE LEVEL

There is an important difference between the way the operating system machine level is supported and the way the conventional machine level is supported. It is due to the fact that the operating system machine level has gradually evolved out of the conventional machine level. Most of the operating system machine level's instructions

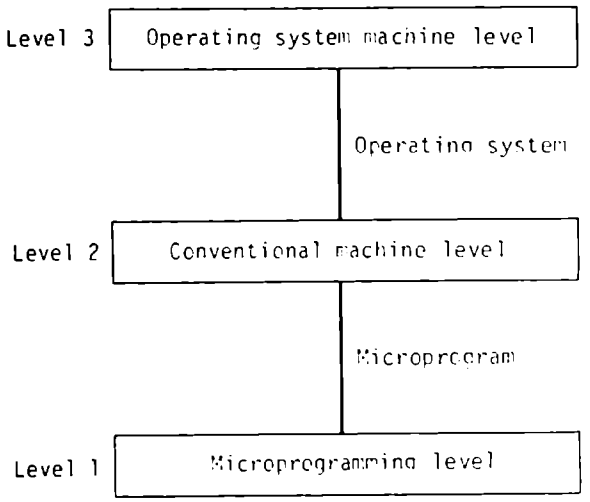


Fig. 5-1. The software supporting the level 3 machine is called an operating system.

are also present at the conventional machine level. We will call these instructions the "ordinary" level 3 instructions, since they include such ordinary operations as arithmetic, boolean operations, and shifting. We will call the other level 3 instructions (those that are not present at level 2) the OSML instructions in order to emphasize their existence only at the Operating System Machine Level.

Although it would be possible to have the operating system interpret all level 3 instructions, doing so is neither efficient nor necessary. The ordinary level 3 instructions can be interpreted directly by the microprogram. This situation is illustrated in Fig. 5-2(a) for the case of a computer with a single memory used to store all programs. As long as only ordinary instructions are being executed, the microprogram fetches instructions directly from the user program, examines them, and executes them.

However, as soon as an OSML instruction is encountered, the situation changes. The microprogram stops interpreting the user program and begins interpreting the operating system instead. The operating system then examines the OSML instruction in the user program and carries it out. When the OSML instruction has been executed, the operating system executes a certain instruction that causes the microprogram to continue fetching and executing user program instructions. Of course, if the next user program instruction is also an OSML instruction, the operating system will be started up again.

This method of executing level 3 programs means that part of the time the computer is functioning as a three-level system and part of the time as a two-level system. During the execution of an OSML instruction, three programs are running—in other words, three processes, one on each (virtual) machine. Each process has its own state, including its own program counter. Conceptually, the level 3 program counter points to the OSML instruction (or its successor, depending on when the program counter

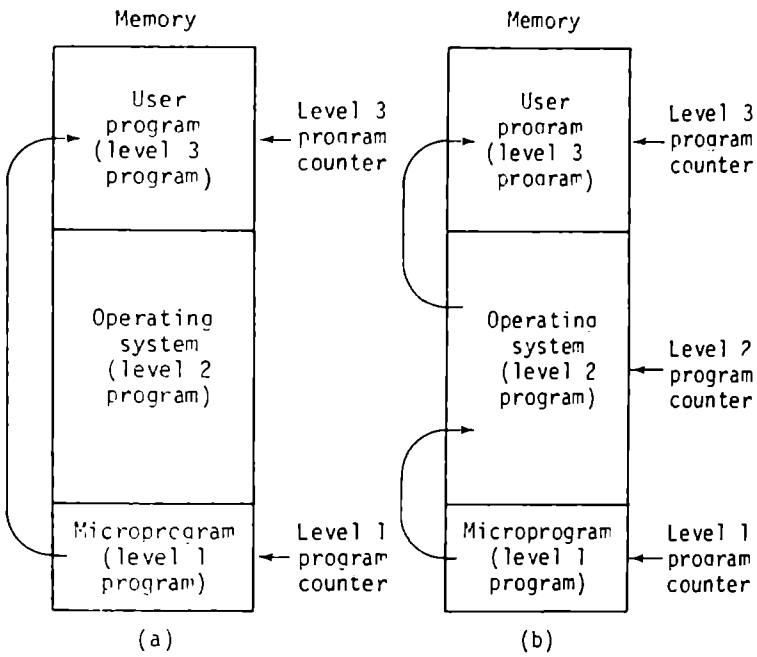


Fig. 5-2. (a) For ordinary instructions, the microprogram interprets the user program directly. (b) For OSML instructions, the microprogram interprets the operating system, and the operating system interprets the OSML instructions.

is advanced), the level 2 program counter points to that operating system instruction currently executing, and the level 1 program counter tells the actual hardware which microinstruction is to be executed.

In this chapter the operating system will be treated as a single level, for simplicity —which does not mean that all operating systems are organized as one level. On the contrary, some advanced operating systems are constructed as a series of several levels. However, the subject of how to design an operating system is beyond the scope of this book. Thus a thorough discussion of how an operating system is actually put together would be out of place here. The reader interested in pursuing the subject of multilevel operating systems should read the article on the subject by Dijkstra (1968b).

It should be mentioned that most operating systems for large computers are **multiprogramming systems**, which means that rather than supporting only one level 3 virtual machine, the operating system supports several level 3 virtual machines in parallel. If each of these virtual machines is connected to a remote terminal, it is called a time-sharing system. If there are no remote terminals, it is called a batch multiprogramming system. Hybrid forms in which some virtual machines are being used on-line, and others are not, are common. A substantial part of the operating system is concerned with managing all the virtual machines rather than with interpreting OSML instructions.

5.2. VIRTUAL I/O INSTRUCTIONS

Normally the level 2 instruction set is completely different from the level 1 instruction set. Both the operations that can be performed and the formats for the instructions are quite different at the two levels. The existence of a few instructions that are the same at both levels is essentially accidental.

In contrast, the level 3 instruction set contains most of the level 2 instructions, with a few new, but important, instructions added and a few potentially damaging instructions removed. Input/output is one of the areas where level 2 and level 3 machines differ considerably. There are two reasons. If a user could execute the real level 2 instructions, he could read confidential data stored in the system that he was not authorized to read, he could write on other users' terminals, and, in general, he could make a big nuisance of himself. Second, no one in his right mind would want to write his own level 2 i/o programs himself.

As an example of the complexity of level 2 i/o, consider the PDP-11, whose level 2 i/o is extremely simple and straightforward compared to most computers. A PDP-11 programmer who wants to do disk i/o at level 2 must consider the possibility that, as a result of the previous disk command, any one of the following 12 errors has occurred, and he must write procedures to handle each case:

1. An i/o operation overflowed out of the last cylinder.
2. An attempt to write on a protected cylinder was aborted.
3. Disk arm failed to seek properly.
4. Error in previous command format.
5. Nonexistent memory specified as buffer.
6. Disk i/o started before previous command finished.
7. Read timing error.
8. Nonexistent disk addressed.
9. Nonexistent cylinder addressed.
10. Nonexistent sector addressed.
11. Checksum error on read.
12. Write check error after write operation.

When one of these errors occurs, the corresponding bit in the error register is set [see Fig. 3-35(b)], along with a bit in the control status register to indicate that an error has occurred. Furthermore, the level 2 programmer must be aware of 43 fields of status information and a total of six disk registers. Few users want to be bothered keeping track of so much information.

5.2.1. Sequential Files

One way of organizing the virtual i/o is to conceive of data that is to be read or written as a sequence of logical records, where a **logical record** is some unit of information meaningful to the programmer. In the simplest case, a logical record might be a single character or an integer. For another application, a logical record might be a 10 × 10 matrix. For still another application, it might be a data structure consisting of five items: 2 character strings, "name" and "supervisor"; 2 integers, "department" and "office"; and a one-bit string, "sex." A sequence of logical records is called a **file**.

The records of a file need not all be of the same length, in which case they are called variable-length records. We will first discuss the concepts of virtual i/o instructions as applied to fixed-length records and then we will treat some existing computers on which variable-length logical records are allowed.

The basic virtual input instruction reads the next record from the specified file and puts it into consecutive cells in main memory, beginning at a specified address, as shown in Fig. 5-3. To perform this operation, the virtual instruction must directly or indirectly specify (at least) two items of information:

1. The file to be read.
2. The main memory address into which the record is to be put.

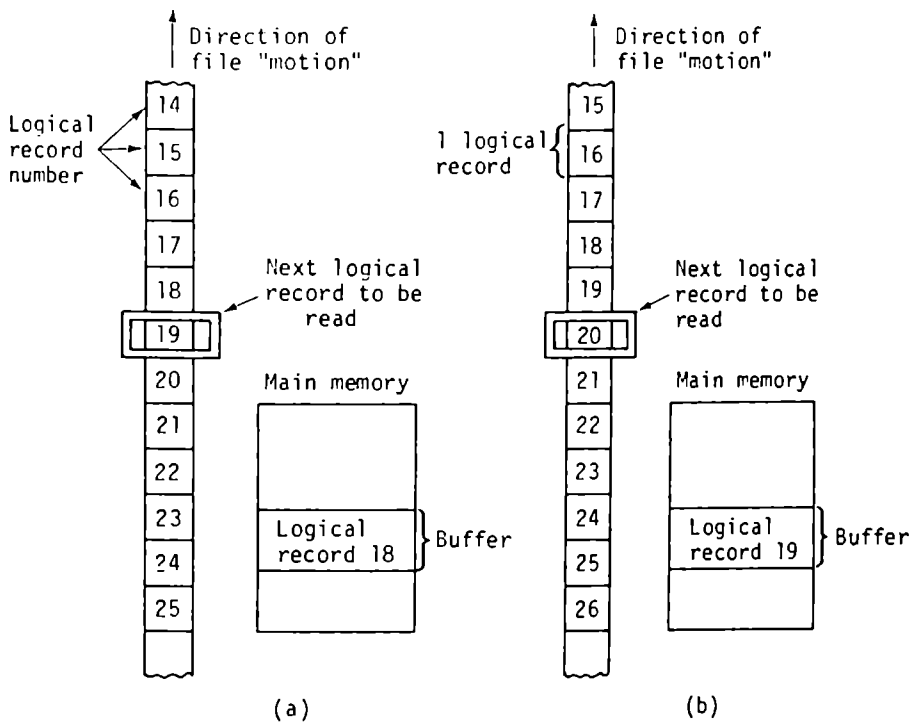


Fig. 5-3. Reading a sequential file. (a) Before logical record 19 is read. (b) After logical record 19 is read.

No "file address" is given. Consecutive sequential READ instructions get consecutive logical records from the file. This situation is in contrast to random access files in the next section, in which the virtual instruction also specifies which logical record is to be read.

The basic virtual output instruction writes a logical record from memory onto a file. Consecutive sequential WRITE instructions produce consecutive logical records on the file. There is also a virtual REWIND instruction that repositions the file at the beginning so that the next logical record read will be the first one. The usual sequence is for a program first to create a file by writing a series of logical records onto it. The file is then rewound and read back one record at a time. In this manner, a file can be used to store large quantities of information that are too big for the main memory. In addition, if the file is stored on a magnetic tape or removable disk pack, it can be transported to another computer and read there.

Many operating systems require that, before a file can be used, it must first be opened; an instruction is provided in such a case. Then the file can be read or written. When the user is finished with a file, he must close it; an instruction is also provided for this operation.

Certain files can be permanently assigned to particular i/o devices. For example, there might be a file called OUTPUT, which consists of a series of 132 character strings and which is associated with the printer. To print a line, the level 3 program writes a 132 character string onto the file output, and somehow or other that string later appears printed on the printed output. The details of how that happens and how the printer actually works are of no concern to the level 3 programmer, although they are, of course, of great concern to the level 2 programmers who must write the software that carries out the virtual instructions (i.e., the operating system writers).

Another example might be a file called INPUT, which consists of 80 character strings. Whenever a READ from input is performed, the contents of the next card are copied to memory. As far as the level 3 programmer is concerned, every instruction to read from input causes the next card in his deck to be read, although, in fact, the entire card deck may have been read all at once some time ago and kept on the disk until needed, at which point one card at a time is copied to the user's buffer in main memory.

5.2.2. Random Access Files

The sequential files discussed previously are not addressable. A virtual READ instruction simply reads the next logical record. The program need not supply the logical record number. Many i/o devices are sequential by nature, such as card readers. The level 3 instruction to read from the file associated with the card reader reads the next card. The program cannot say "Now read the 427th card" unless, by accident, it happens to have already read the first 426 cards. A sequential file is therefore a good model of such a device.

For some applications, the program needs to access the records of a file in a different order than the order in which they have been written. As an example,

consider an airline reservation system in which the passenger list for each flight forms a logical record and all the flights for one day form a file. A person may call up the airline's ticket agent and inquire about the availability of a seat on a flight to White Plains next Wednesday. The ticket agent enters the question into an on-line terminal. If the passenger list for that flight is record 26 of a certain file, the program needs only record 26. It clearly should not have to read sequentially, starting at record 1 until it gets to record 26. There is a need for the program to be able to access a specific record from the middle of a file by giving its record number.

Similarly, it is sometimes necessary to rewrite a particular logical record on a file without rewriting any logical records before or after it. In the preceding example the person might want to make a reservation on the flight he inquired about. In order for this action to be performed, the program must rewrite the logical record containing the reservation list, adding the caller's name to the passenger list. It is neither necessary nor desirable to change any other logical records, however.

Most operating systems provide a virtual instruction to read the Nth logical record of a file. These virtual instructions must provide (at least) three items of information:

1. The file to be read.
2. The main memory address into which the record is to be put.
3. The position of the logical record within the file.

The corresponding WRITE instructions must also provide this information as well.

Another form of file organization is one in which logical records are addressed not by their position in the file but by the contents of some field within each logical record, called the **key**. For example, a file containing a company's personnel data will have one field in each record containing the employee's name. A virtual instruction might be provided that allowed the program to give the name of an employee and have his logical record read in. It is the responsibility of the operating system to search the file for the needed logical record, saving the programmer the effort necessary to write the search procedure himself. This situation is analogous to a level 2 multiplication instruction, which spares the programmer the effort of microprogramming his own multiplication procedure.

On some computers, a distinction is made between files that may be addressed by record number or key and files on which only the next record may be read. The former are called **random access files**, to distinguish them from the latter, the **sequential files**. On other computers, no such distinction is made, and both kinds of virtual instructions (with and without addressing) are allowed on all files.

5.2.3. Implementation of Virtual I/O Instructions

In understanding how virtual i/o instructions are implemented on the level 2 machine, one must examine how files are organized and stored. In the following discussion we will assume that a disk is being used to hold the files, but similar considerations apply to other media.

A basic issue that must be dealt with by all file systems is allocation of storage. A disk consists of a series of arm positions or cylinders, each of which has one or more tracks, equal to the number of surfaces (typically, 2 to 20). Tracks are divided into sectors, each of which holds a certain number of words. On some disks, the sector size is adjustable. For example, the programmer may choose to format a track as 10 sectors of 600 bytes, 12 sectors of 500 bytes, or 15 sectors of 400 bytes. On other disks, the sector size is fixed.

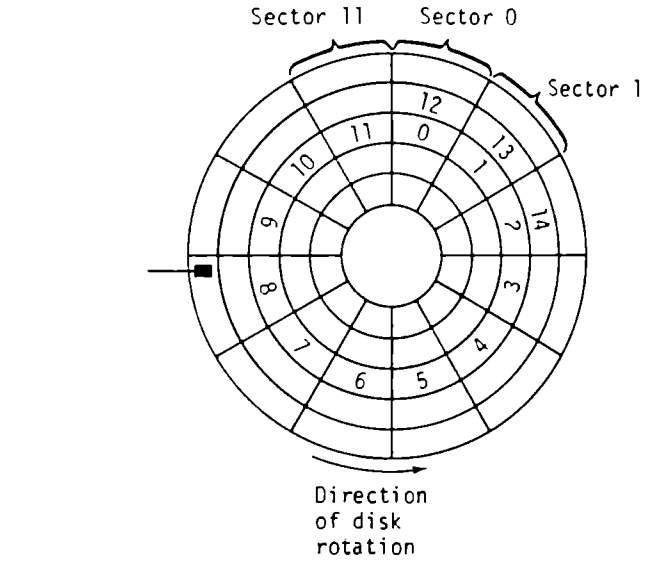
A fundamental property of a file system implementation is the size of the unit in which space is allocated. A disk has three reasonable candidates: the sector, the track, or the cylinder. Allocating space in units of 2.93 tracks would be absurd. The difference can be most clearly seen in the case of a file consisting initially of only a single character. If the sector is the allocation unit, only one sector will be reserved for the file, and the other sectors on the same track will be available for use by other files. If the track is the allocation unit, one entire track will be reserved for the file, but the other tracks in the same cylinder will be available for other files. If disk space is allocated by the cylinder, an entire cylinder will be reserved even for a file of only one character.

Another fundamental property of a file system implementation is whether a file is stored in consecutive allocation units or not. Figure 5-4 depicts a simple disk with one surface consisting of five tracks of 12 sectors each. Figure 5-4(a) shows an allocation scheme in which the sector is the basic unit of space allocation and in which a file consists of consecutive sectors. Figure 5-4(b) shows an allocation scheme in which the sector is the basic allocation unit, but in which a file need not occupy consecutive sectors.

If the track is the allocation unit, then a consecutively allocated file will occupy consecutive tracks. As a rule, all the tracks in a cylinder will be allocated before the next cylinder is allocated. If a file is allocated in units of tracks but is not consecutively allocated, then the tracks may be chosen anywhere on the disk, without regard to proximity to one another.

There is an important distinction between the level 3 view of a file and the operating system's view of a file. The level 3 program sees the file as a linear sequence of logical records, card images, print lines, arrays, and so on. The operating system sees the file as an ordered, although not necessarily consecutive, collection of allocation units.

In general, the logical record size will differ from the allocation unit size, perhaps being smaller or perhaps being larger. A file may consist of a sequence of 80-byte strings stored on a disk on which space is allocated in units of 6144-byte tracks. Bytes 0 to 79 of the first track will contain the first logical record, bytes 80 to 159 will contain the second logical record, and so forth. The tracks are regarded as logically contiguous, even if they are not physically contiguous, and a logical record may be split over two tracks. It is the task of the operating system to make the physical-allocation unit size transparent to the level 3 program. When the level 3 program asks for logical record N, it gets logical record N, without regard to which track or tracks that record may occupy.



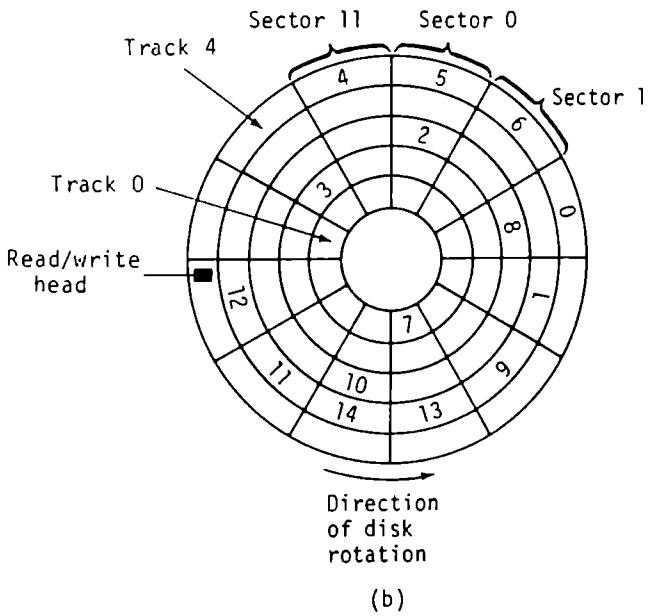


Fig. 5-4. Different disk allocation strategies. (a) The file is stored in consecutive sectors. (b) The file is not stored in consecutive sectors.

In order for the operating system to deliver logical record N of some file on request, it must have some method for locating it. If the file is allocated consecutively, the operating system need only know the location of the start of the file and the sizes of the logical and physical records in order to calculate the position of the logical record. For example, if a logical record consists of eight words and a track consists of 100 sectors of 64 words apiece, logical record 5000 will be in sector 26 of track 7. From its knowledge of the location of the first track, the level 2 software can calculate the exact disk address of the needed sector and issue a disk command to read it.

If the file is not allocated consecutively, it is not possible to calculate the position of an arbitrary logical record from the position of the start of the file alone. In order to locate any arbitrary logical record, a table called a **file index**, giving the allocation units and their actual disk addresses, is needed. The file index can be organized either in terms of the logical records, giving the disk address of each one, or simply as a list of the allocation units and their disk addresses. To illustrate how the file index is used, consider a disk allocation in sectors, as in Fig. 5-4(b), with 512 bytes per sector. The file consists of 132-byte logical records (print lines) with bytes 0 to 131 constituting the first logical record, bytes 132 to 263 the second logical record, and so on. Logical record 21 occupies bytes 2640 to 2731, which are in the sixth allocated sector. By using the file index, one can look up the address of the required sector, no matter which file index organization is used.

An alternate method of locating the allocation units of a file is to organize the file as a linked list. Each allocation unit contains the address of its successor. This can be done most efficiently if the hardware provides an extra word in each allocation unit for storing this address. The method is equivalent to dispersing the file index throughout the file. On a disk file allocated by sectors, each sector would contain the address of the succeeding sector. Such a file can only be read sequentially and cannot be accessed randomly.

Up until now we have discussed both consecutively allocated files and nonconsecutively allocated files, but we have not specified why both kinds are used. When a user creates a file, sometimes he knows the maximum size that the file will later attain and sometimes he does not. As an example of the latter, consider the computerized accounting system of the Ecology Manufacturing Company, which has just announced their latest product, a wind-up toothbrush. Their computer has a file listing the names and addresses of all their customers; but at the time the wind-up toothbrush is put on sale, they have no idea how many customers they will eventually have and, consequently, do not know how large the customer file will eventually become.

When the maximum file size is not known in advance, it is usually impossible to use a consecutively allocated file. If the file is started at track I and allowed to grow into consecutive tracks, it may bump into another file at track J and have no room to expand. If the file is not allocated consecutively, this situation presents no problem, since succeeding tracks can be put in any available cylinder. If a disk contains a number of "growing" files, none of whose final sizes is known, storing each of them

as a consecutive file will be impossible. Moving an existing file is sometimes possible but always expensive.

If the maximum file size is known in advance, a region of the disk can be allocated when the file is created, even if the data is not yet available. For example, daily weather data for 1977 will require 365 records and can be allocated on or before December 31, 1976, even though none of the data is known at file creation time. Consecutively allocated files are less flexible than nonconsecutively allocated files, because their maximum sizes must be known in advance; on the other hand, they are simpler to implement because no file index is needed. Note that both consecutively and nonconsecutively allocated files can be used as sequential access and as random access files.

In order to allocate space on the disk for a file, the operating system must keep track of which allocation units are available and which are already in use storing other files. One method consists of maintaining a list of all the holes, a hole being any number of contiguous allocation units. Figure 5-5(a) illustrates this list for the disk of Fig. 5-4(b).

An alternate method is to maintain a bit map, with one bit per allocation unit, as shown in Fig. 5-5(b). A one bit indicates that the allocation unit is already occupied and a zero bit indicates that it is available.

Track	Sector	Number of sectors in hole
0	0	5
0	6	6
1	0	10
1	11	1
2	1	1
2	3	3
2	7	5
3	0	3
3	10	3
4	3	8

(a)

Track \ Sector	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	0	1	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	1	0
2	1	0	1	0	0	0	1	0	0	0	0	0
3	0	0	0	1	1	1	1	1	1	0	0	0
4	1	1	1	0	0	0	0	0	0	0	0	1

(b)

Fig. 5-5. Two ways of keeping track of the available disk sectors. (a) A list of holes. (b) A bit map. A 0 indicates an available sector.

The first method has the advantage of making it easy to find a hole of a particular length, but it has the disadvantage that the table must be sorted on hole length to permit quick lookups. It also has the disadvantage of being variable sized. As files are created and destroyed, the length of the list will fluctuate, an undesirable characteristic. The bit table has the advantage of being constant in size. In addition, changing the status of an allocation unit from available to occupied is just a matter of changing one bit. Both methods require that when any file on the disk is allocated or returned, the allocation list or table be updated.

Before leaving the subject of file system implementation, it is worth commenting about the size of the allocation unit. Very few files will occupy exactly an integral number of allocation units. Therefore some space will be wasted in the last allocation unit of nearly every file. If the file is much larger than the allocation unit, the average space wasted will be half of an allocation unit. The larger the allocation unit, the larger the amount of wasted space.

If the expected file size is short, it is inefficient to allocate disk space in large units. For example, if the majority of users of a proposed file system will be students with short programs averaging about 3000 characters, and a disk track contains 100 sectors of 640 characters apiece (i.e., 64,000 characters), it would be foolish to allocate space in units of a track or, worse yet, a cylinder.

A disadvantage of allocating space in small chunks is that the file index and bit map (if used) will be large. Furthermore, if the file is nonconsecutively allocated, it will, in general, be necessary to do one disk seek per allocation unit. Disk seeks are slow; having to seek every 640 characters is less desirable than having to seek every 64,000 characters. Of course, the strategy used to allocate new space to growing files can take this into consideration and try to assign allocation units close to the existing ones.

Few computers have virtual i/o instructions as simple and straightforward as the conceptual i/o instructions discussed in the previous sections. In the following sections we will examine how the concept of virtual i/o instructions have been implemented on three commercially available computers—the IBM 370, Cyber 70, and PDP-11.

The virtual i/o instructions on the 370, Cyber 70, and PDP-11 differ enormously in their power and complexity. The 370 provides a large variety of file types, access methods, and virtual instructions, which combined are capable of performing a wide class of i/o operations. Correspondingly, learning how to use all these features is a task requiring months. The Cyber 70 virtual i/o and file system are somewhat simpler. In contrast to the 370 and Cyber 70, the PDP-11 virtual i/o instructions are quite straightforward, but, of course, they do far less than the 370's.

The sections following are simply brief sketches of some of the interesting features. Readers intending to use any of the virtual i/o instructions should consult the appropriate manufacturer's manuals.

5.2.4. IBM 370 Virtual I/O

On the 370, files are distinguished by the method in which the data is accessed. We will mention just four kinds: sequential files, partitioned files, indexed sequential files, and direct access files. A **sequential** file is one in which the logical records will be read in order, one after another. This is the only kind allowed for files stored on sequential devices, such as magnetic tapes. A **partitioned** file consists of several parts, each of which is a sequential file. In a partitioned file, each of the parts has a name, and the program may use the name to access a part. Within a part, all accesses are sequential. This file type is often used for program libraries.

Files in which the logical records are requested on the basis of a key are often

useful. For instance, consider a hypothetical automated zoo with a terminal in front of each cage on which visitors can ask questions about the habits of the animals. The information could be organized as a single file, with one logical record per species. When a visitor asked for information about giant pandas, the program would have to find the logical record containing the information about pandas. An **indexed sequential** file could be used here.

Associated with each indexed sequential file is an index containing one entry for each logical record in the file. Each entry contains a character string identifying the record, along with the record's location in secondary memory (e.g., disk address). Such an index is illustrated in Fig. 5-6. In this example the animal species would be the keys. A virtual i/o instruction for accessing indexed sequential files provides a key, which the operating system looks up in the index in order to find the logical record. Because a search must be performed, this access method can be slow if a large number of logical records exist.

Record key	Cylinder	Track	Sector
California condor	4	0	10
European bison	16	1	3
Giant panda	9	1	4
Monkey-eating eagle	34	0	11
Orangutang	84	1	3
Pelican	19	1	7
Whooping crane	16	0	2
Wild yak	5	1	5

Fig. 5-6. An index for an indexed sequential file.

Figure 5-6 is a conceptual representation of an index, rather than an attempt to depict an actual 370 index. There are, in reality, three different kinds of indices—track indices, cylinder indices, and master indices, which form an index hierarchy. Furthermore, a distinction is made between data in the "prime" tracks and data in the "overflow" tracks. When a file is created, all the data is put in the prime area. If new records are added later, there may be insufficient room in the appropriate prime tracks, so they are put in the overflow tracks. Moreover, Fig. 5-6 assumes that the logical records are unblocked (not packed together into larger units for reading and writing efficiency), although files on the 370 may be blocked or unblocked.

A **direct access** file is one in which the virtual i/o instruction explicitly specifies the actual disk address where the logical record is to be found. This method is the fastest, but it requires the most work from the programmer, for he must maintain an index himself. A variation of this file access method allows a logical record to be specified by its position within the file—that is, by its logical record number.

Associated with each file is a table, which on the 370 is called a data control block or DCB. In order for a file to be read or written, it must have a DCB in main memory. Figure 5-7 shows a few of the many fields in a DCB.

```
Specification of the file name
Maximum logical record length
Buffer size
Number of buffers required
Access method
Record type (variable length/fixed length)
Key length
Key position within logical record
Address of end-of-file procedure
Address of error handling procedure
Number of overflow tracks per cylinder
Other information
```

Fig. 5-7. Some of the fields required for the DCB of an indexed sequential file on the 370.

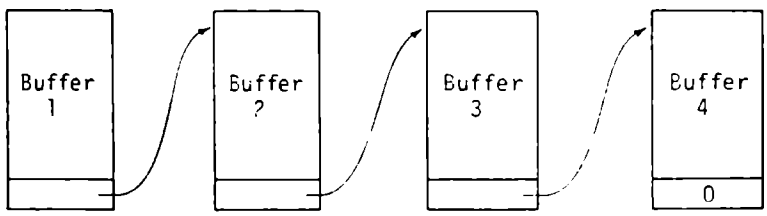


Fig. 5-8. Anticipatory buffering provides for automatic prefilling of buffers, before they are needed. While the program is using buffer 1, the operating system can be busy filling buffers 2, 3, and 4.

When a file is being read sequentially, the operating system knows which logical record will be needed next—namely, the one following the logical record most recently read. To speed up i/o, the operating system can read one or more logical records in advance, so that when the program executes a virtual READ instruction, the next record will already be in main memory and no time will be lost waiting for it. This technique is called **queueing** or anticipatory buffering and is shown in Fig. 5-8. For files accessed randomly, queueing is not possible because the operating system cannot predict which logical record will be needed next, as it can with sequential access.

Anticipatory buffering on sequential files is called Queued Sequential Access Method or QSAM. Indexed sequential files may also be read sequentially instead of randomly. When an indexed sequential file is being read sequentially, queueing is possible and is called Queued Indexed Sequential Access Method or QISAM. Queueing is not permitted on partitioned or direct files. The names of some access methods are given in Fig. 5-9.

Separate virtual i/o instructions are provided for the queued and basic (nonqueued) access methods. Queued access is performed by the virtual instruction

```
GET dcbname, buffer
```

where dcbname is the name of a DCB and buffer is the address where the logical record is to be put. If the logical record has already been read, it is copied into the

File type	With anticipatory buffering	Without anticipatory buffering
Sequential	QSAM—Queued sequential access method	BSAM—Basic sequential access method
Indexed sequential	QISAM—Queued indexed sequential access method	BISAM—Basic indexed sequential access method
Partitioned	Not allowed	BPAM—Basic partitioned access method
Direct	Not allowed	BDAM—Basic direct access method

Fig. 5-9. Some of the file access methods available on the 370.

buffer. If it has not been read, the program will have to wait until it has been read. The farther ahead the system reads in advance, the less likely it is that the program will have to wait for a GET to be completed. Of course, if the operating system tries to keep 10 records ahead of the program, a lot more memory will be needed for buffer space than if it only tries to keep one record ahead. The programmer specifies how far ahead the system should read by a field in the DCB.

Similarly, output can be queued. With queued output, the program can fill up a new buffer (or buffers) even though the previous one has not yet been output. With basic output (e.g., BSAM), the program must wait until the output of a buffer is complete before continuing. The virtual instruction for queued output is

```
PUT dcbname, buffer
```

For unqueued input and output (i.e., BSAM, BISAM, BPAM, or BDAM), the virtual instructions are READ and WRITE instead of GET and PUT.

The GET, PUT, READ, and WRITE virtual instructions, as well as all other virtual instructions on the 370, are mnemonics for supervisor calls (SVCs). An SVC tells the microprogram to stop interpreting the user program and start interpreting the operating system. The operating system then examines and interprets the virtual instruction.

5.2.5. Cyber 70 Virtual I/O

In this section we will discuss how the concept of virtual i/o instructions has been implemented on the Cyber 70. Virtual i/o instructions on the Cyber 70 use two tables, a file environment table (FET) and a circular buffer, as shown in Fig. 5-10. Virtual READ instructions read data into the circular buffer, and virtual WRITE instructions write data from the circular buffer. The FET describes the status of the circular buffer.

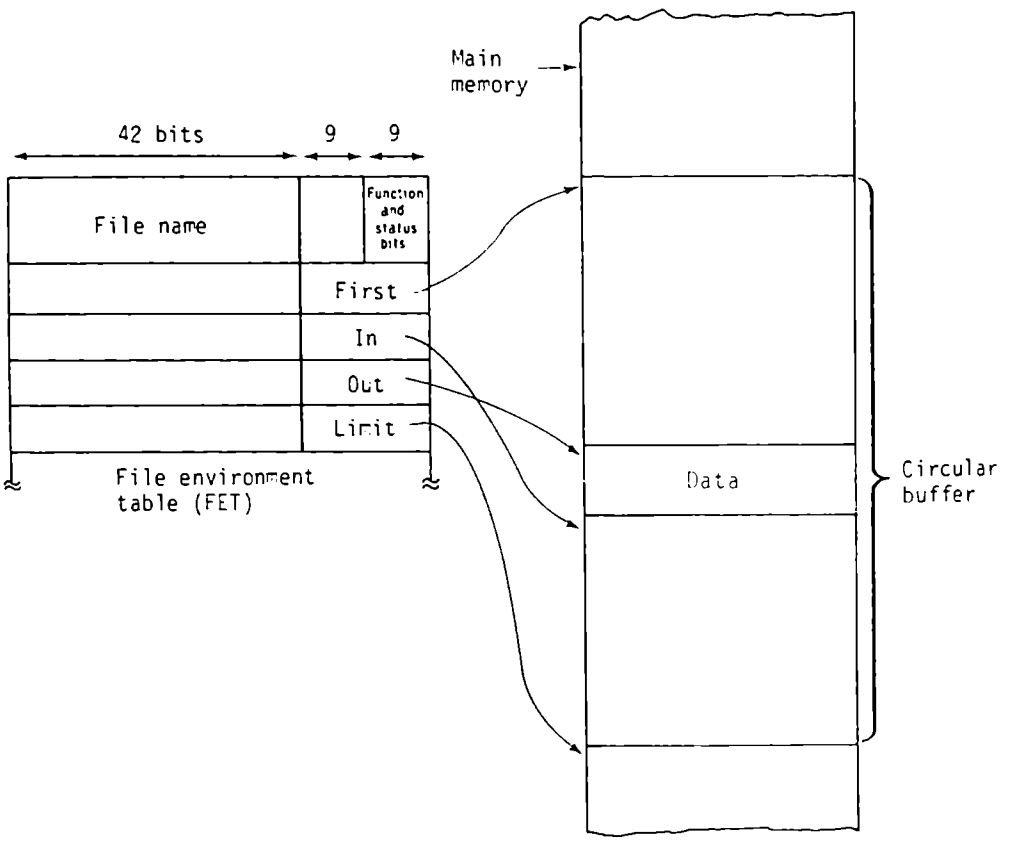


Fig. 5-10. Virtual i/o instructions on the Cyber 70 use an FET and a circular buffer.

An FET may contain more than 50 fields of information, only six of which are shown in Fig. 5-10. The Cyber 70's FET is analogous to the 370's DCB.

The fields FIRST, IN, OUT, and LIMIT are pointers to the circular buffer used for the file named in the FET. FIRST points to the first word of the buffer, and LIMIT points to the first word beyond the buffer—that is, the last word plus one. The buffer is regarded as the surface of a cylinder, with the last word being followed directly by the first word. Figure 5-11 illustrates the circular nature of a buffer with 65 words. When 10 words are placed in the circular buffer, starting at word 60, they occupy words 60, 61, 62, 63, 64, 0, 1, 2, 3, and 4 as shown. In other words, the buffer "wraps around," hence the name circular buffer.

The pointers IN and OUT have different meanings, depending on whether the file is being read or written. If the file is being read, IN points to the word in the buffer where the next data item read will be placed. After a word is put in the buffer, IN is increased by 1. After IN is increased, a test is made to see if IN = LIMIT, and if so, IN is reset to FIRST to achieve the circular buffer effect. When the program needs

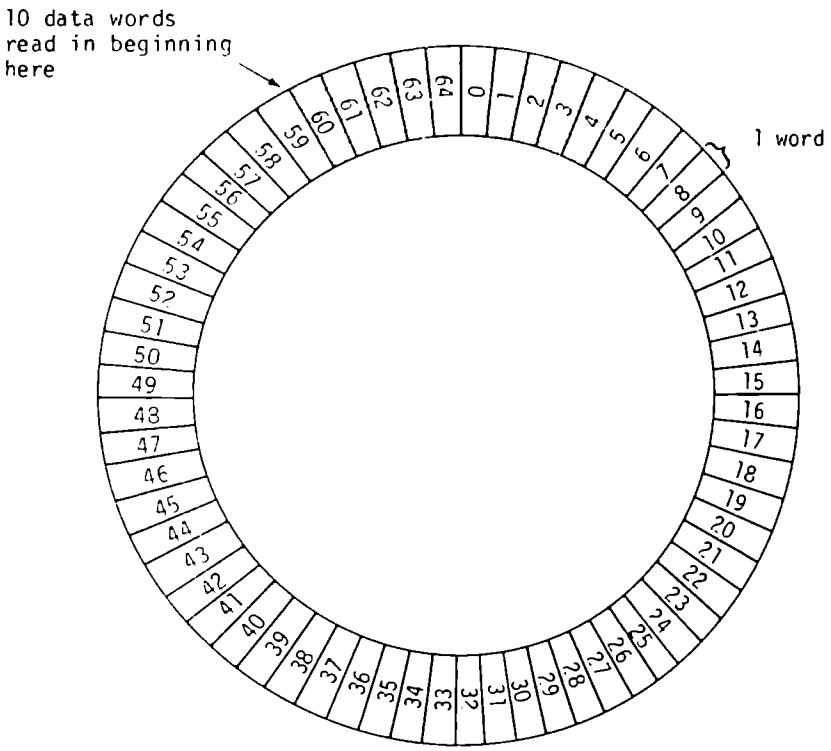


Fig. 5-11. When ten words are put into a circular buffer beginning at word 60, they fill the remainder of the buffer and wrap around, continuing at 0.

a word of data from the file, it takes the word pointed to by OUT and then increments OUT by 1, resetting OUT to FIRST if OUT = LIMIT. Figure 5-12 illustrates the use of a circular buffer.

When an FET is created, IN = FIRST and OUT = FIRST. Whenever IN = OUT, the buffer is empty. After the first virtual READ instruction is complete, the situation is as shown in Fig. 5-12(b), with IN pointing to the word following the last data word. OUT is unchanged. Now the program begins removing data from the buffer to use it. The word pointed to by OUT is removed and OUT is incremented by 1. Then another word is removed and OUT is incremented again. After part, but not all, of the data has been removed from the buffer by the CPU program, the situation is as shown in Fig. 5-12(c).

At this point the program can either continue removing data from the buffer until IN = OUT, meaning the buffer is empty, or it can issue another virtual READ instruction to continue filling the buffer, starting at IN. I/O can proceed simultaneously with the program; so if the program removes data at approximately the same rate as it is put into the buffer, the program can run indefinitely without having to

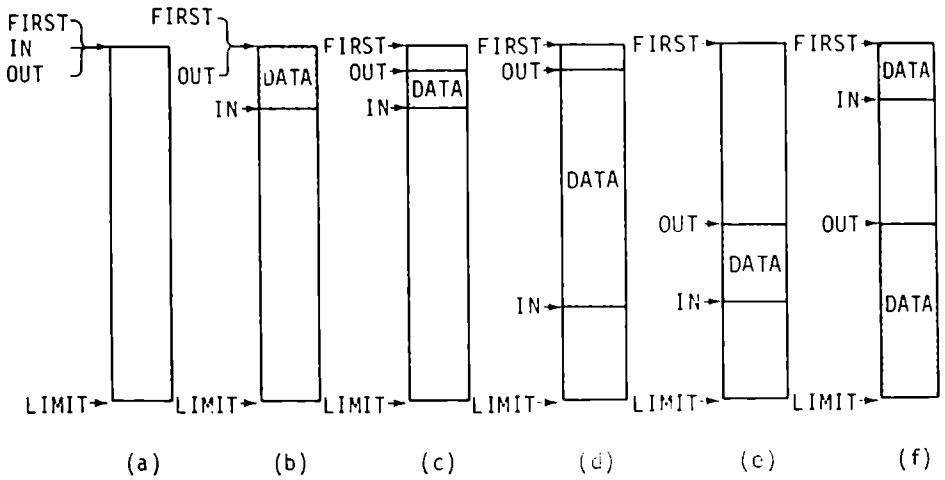


Fig. 5-12. Use of a circular buffer.

wait for more data. It is also possible to specify another level of buffering by using ECS. If this is done, the main memory buffer is filled from ECS (at up to 100 characters per microsecond) and the ECS buffer is filled from the disk.

Of course, if the program removes data too quickly, the buffer will become empty. If the CPU removes data too slowly, the buffer will fill up and i/o will cease. If IN = OUT − 1 (or IN = LIMIT −1 and OUT = FIRST), then i/o must stop. If one more word were read in, IN would become equal to OUT, indicating an empty buffer. Consequently, a circular buffer can never become completely full. There must always be at least one empty word; and in order to accommodate this empty word, the buffer must be at least one word larger than the unit in which data is transferred.

The Cyber 70 has a standard physical record length for each i/o device and parity. The physical record length is the unit in which data is normally read or written. For disk files, the physical record length is 64 60-bit words (1 sector). For binary (odd-parity) tapes, it is 512 words. For tapes containing character data (even parity), it is 128 words.

A logical record consists of zero or more full physical records, followed by one physical record that is not full. In Fig. 5-13(a), a logical record consists of 3.5 physical records (sectors, for disk files). Each logical record begins at the start of a new sector. In Fig. 5-13(b), the logical records are shorter than a sector, so each sector contains a logical record. If, by accident, a logical record happens to be an integral number of physical records, a zero-length physical record must follow it, to indicate the end of the logical record. For disk files, this means that an empty sector must follow any logical record of length 64, 128, 192, etc. words. The number of words in a disk sector is recorded in the sector itself, in addition to the data words.

When a program wants to execute any virtual instruction, including a virtual i/o instruction, it puts a message in word 1 of its portion of the main memory. These messages all consist of three characters in the leftmost 18 bits, specifying the opcode,

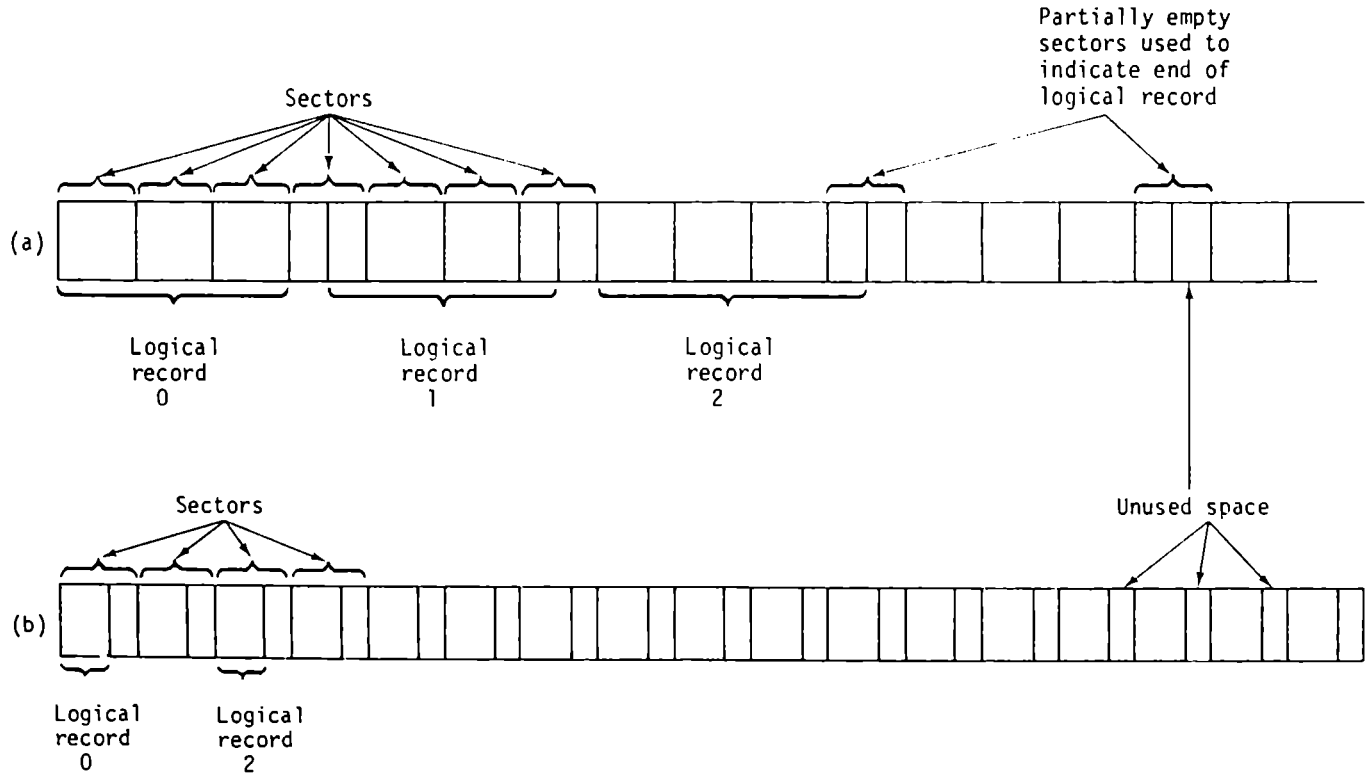


Fig. 5-13. (a) Cyber 70 logical records longer than a sector. (b) Cyber 70 logical records shorter than a sector.

and frequently other information as well. Essentially, they are 60-bit virtual instructions. Figure 5-14 shows the CIO instruction, used for virtual i/o. The rightmost 18 bits contain the FET address, from which the operating system can find the file name, the function bits, and the buffer parameters. The function bits specify the i/o operation desired on the file—for example, read, write, rewind, skip, backspace, write end-of-file, unload, and a number of others.

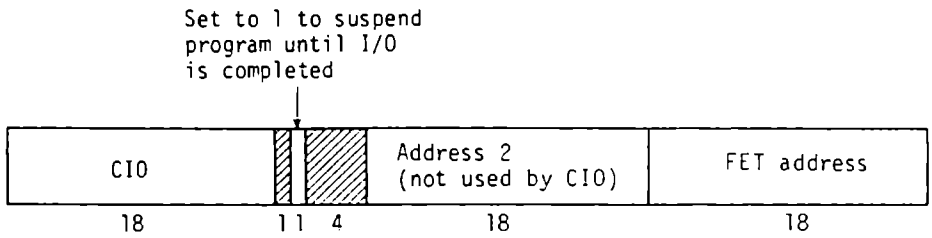


Fig. 5-14. Format of the Cyber 70 OSML instructions. CIO is the opcode for Circular buffer Input Output, with the FET providing the specific function.

Each three-letter virtual opcode is, in fact, the name of a PPU program. One of the PPUs periodically examines word 1 of each program in main memory. Whenever it finds one of these words nonzero, it fetches the 18-bit virtual opcode and checks it for validity. If it is the name of a PPU program, that PPU program is fetched from the PPU library, loaded into any available PPU, and started. The PPU program then examines the rest of the virtual instruction and carries it out. This method of executing virtual instructions makes it very easy for CDC to add new virtual instructions to the Cyber 70. They need only add new PPU programs to the library.

Like the 370, the Cyber 70 has several different virtual instructions for reading and writing files. For example,

```
READ file
```

causes one logical record from the specified file to be read into the buffer pointed to by the FET for that file. If there is insufficient room in the circular buffer for the next logical record, only a portion of the next logical record will be read. In contrast to the virtual instruction READ, which only reads one logical record, the virtual instruction READNS continues reading until the circular buffer is nearly full. Reading stops when the amount of space remaining is insufficient for another physical record.

There is also a set of virtual instructions for reading and writing random access files. The method used is similar to the indexed sequential files on the IBM 370. A file can be written with the virtual instruction WRITOUT, which makes an entry in an index for each logical record. The entries may be either numbers or keys. A READIN virtual instruction can be used to read a specific logical record whose number or key (character string) is given. Virtual instructions are also provided for various other functions, including rewinding a file, skipping forward or backward,

and writing an end-of-file mark, used to indicate the end of a file. These virtual instructions all use CIO, with different function bits in the FET.

5.2.6. PDP-11 Virtual I/O

The PDP-11 Disk Operating System (DOS-11) provides virtual i/o instructions for reading and writing two kinds of files, contiguous and linked. A **contiguous** file consists of a series of consecutive disk sectors. The maximum size of a contiguous file must be specified when the file is created, and it may not be changed subsequently. A **linked** file is stored in sectors not consecutively allocated. The maximum size of a linked file need not be specified when a file is created, and it may change subsequently.

Linked files do not use file indices. Instead each disk sector contains a pointer to its successor. For this reason, it is not possible to find block N without first reading all the preceding blocks. Consequently, random access to a linked file is very costly, because it requires reading the file sequentially until the desired block is found. The random access behavior is accomplished by a technique that is much slower than maintaining a file index in main memory.

Different virtual i/o instructions are provided for the two kinds of files. A virtual instruction to read the file until a line feed is encountered is available for each one. This instruction (.READ) allows the program to read a file consisting of a series of lines, where, by definition, lines are terminated by line feeds. Most programs and data take the form of a sequence of lines, so this virtual instruction has wide applicability. Lines need not be the same length because they are terminated by an explicit character. This point is particularly useful for files typed at an on-line terminal, where, in general, lines have differing lengths, as opposed to punched cards, which all have the same length.

The virtual i/o instruction for reading the next line has two addresses, each of which is the start of a short table. The first table specifies the device on which the file is located, as well as a word that provides information about the file name. The second table contains the address and length of the buffer in which the line is placed after it is read, plus a counter telling how many bytes were read. This table is similar to the FET on the Cyber 70, except that the buffer is not circular. The line is always positioned at the start of the buffer, thus eliminating the need for one of the four FET pointers. The virtual "write a line" instruction (.WRITE) is analogous to the virtual "read a line" instruction, with the line coming from instead of going to the buffer.

A second virtual i/o instruction is "transfer a logical record" (.RECRD), which reads or writes a specific logical record into a buffer or from a buffer independent of the current position of the file. This is a random access virtual instruction. It is permitted only on contiguous files consisting of fixed-length logical records. It is not permitted on linked files because it would be very inefficient. Like the .READ virtual instruction, .RECRD has two addresses. The first points to a table used to find the file name and device. The second points to a table giving the buffer address,

the record length, the number of the record to be transferred, and whether a read or a write is desired.

A third virtual i/o instruction is "transfer a physical record" (.BLOCK), which is only permitted on linked files and not on contiguous files. It is like .RECRD except that a physical record instead of a logical one is transferred. On the disk, for example, a physical record is a sector, and it consists of 256 words. The .BLOCK virtual instruction to read physical record 0 will read words 0 to 255 of the file, disregarding any logical structure in the information. Similarly, .BLOCK with a block number of 5 will read words 1280 to 1535, even on a file consisting of lines, despite the fact that this information may not correspond to an integral number of lines.

In order to execute a virtual instruction, a PDP-11 program executes an emulator trap instruction (EMT). This instruction contains a number from 0 to 255 in the rightmost eight bits of the (16-bit) instruction, specifying which virtual instruction is to be executed. In other words, the low-order byte of the EMT instruction contains the opcode of the virtual instruction. The EMT causes the program counter and PSW to be pushed onto the stack and a new program counter and PSW to be fetched from a trap vector used only by the EMT instruction. The operating system examines the EMT instruction in order to find the virtual opcode and then carries it out.

The PDP-11 EMT instruction is essentially identical to the 370 SVC instruction. Both cause the microprogram to stop interpreting the level 3 program and start interpreting the operating system. The only difference is that on the PDP-11 the old program counter and PSW are pushed onto the stack, whereas on the 370 this information is stored at a fixed location in memory.

5.2.7. Comparison of Level 3 I/O

The virtual i/o instructions on the IBM 370, Cyber 70, and PDP-11 have a number of points in common:

1. Both sequential and random access reading are allowed.
2. The logical record length is user determined.
3. The quantity of data read may be either a logical or a physical record.
4. The virtual instructions use tables containing information about files.
5. The i/o can proceed in parallel with the CPU program.

There are also some differences between them. The Cyber 70 is the only one to use circular buffers. The 370 uses a pool of buffers for each file, whereas the PDP-11 uses only one buffer per file. The 370 is the only one of the three to provide for anticipatory buffering of sequential files, although the Cyber 70 double buffering using ECS gives a similar effect. The PDP-11 has neither facility. Nor does it have indexed sequential files with keys, as do the other two. The PDP-11 is also the only one of the three on which random access by logical record is restricted to files whose maximum size is known when the file is created.

5.3. VIRTUAL INSTRUCTIONS USED IN PARALLEL PROCESSING

Some computations can be most conveniently programmed for two or more cooperating processes, running in parallel (i.e., simultaneously, on different processors), rather than for a single process. Other computations can be divided into pieces, which can then be carried out in parallel to decrease the time required for the total computation. In order for several processes to work together in parallel, certain virtual instructions are needed. These instructions will be discussed in the following sections.

The laws of physics provide yet another reason for the current interest in parallel processing. According to Einstein's special theory of relativity, it is impossible to transmit electrical signals faster than the speed of light, which is nearly 1 ft/nsec. This fact has important implications for computer organization. For example, if a CPU needs data from the main memory 1 ft away, it will take at least one nanosecond for the request to arrive at the memory and a second one for the reply to get back to the CPU. Consequently subnanosecond computers will need to be extremely tiny. An alternate approach to speeding computers up is to build machines with many CPUs. A computer with a thousand 1-nsec CPUs may have the same computing power as one CPU with a cycle time of 0.001 nsec, but the former may be much easier and cheaper to construct.

On a computer with more than one physical processor, each of several cooperating processes can be assigned to its own processor so as to allow the processes to progress simultaneously. If only one physical processor is available, the effect of parallel processing can be simulated by having the processor run each process in turn for a short time. In other words, the processor can be shared among several processes.

Figure 5-15 shows the difference between true parallel processing, with more than

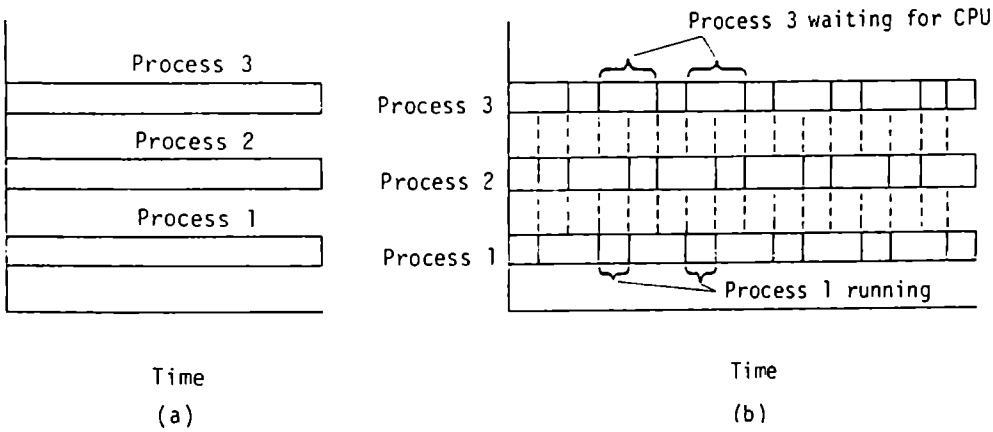


Fig. 5-15. (a) In true parallel processing, each process has its own CPU, and several processes can run simultaneously. (b) In simulated parallel processing, the CPU is rapidly switched from process to process, giving the illusion of parallel processing with slower CPU's.

one physical processor, and simulated parallel processing, with only one physical processor. Even when parallel processing is simulated, it is useful to regard each process as having its own dedicated virtual processor. The same communication problems that arise when there is true parallel processing also arise in the simulated case.

5.3.1. Process Creation and Destruction

When a program is to be executed, it must run as part of some process. This process, like all other processes, is characterized by a state and an address space through which the program and data can be accessed. The state includes the program counter and, possibly, a program status word, a stack pointer, and general registers.

Simple operating systems usually support a fixed number of processes, all of which are created when the computer is started up in the morning and all of which remain in existence until the computer is stopped at night. On these computers, a program must wait in the input queue until a process becomes available before it can be loaded into the process' address space and executed.

More sophisticated operating systems allow processes to be created and destroyed without stopping the computer. A computer with this kind of operating system can support a variable number of level 3 machines, each process corresponding to one virtual machine. In order to take full advantage of parallel processing, a level 3 program needs a virtual instruction to create new processes, to which it can delegate work. Some operating systems provide a level 3 instruction for creating a new process, allowing the creating process to specify the initial state of the new process, including its program, data, and starting address.

The programming language PL/I allows the programmer to write programs using parallel processing, provided that the level 3 machine language to which the PL/I

```
MAIN:       PROCEDURE   OPTIONS (MAIN);
            .
            .
            .
            CALL    PROCESS1    TASK (T1);
            CALL    PROCESS2    TASK (T2);
            .
            .
            .
END MAIN;

PROCESS1:   PROCEDURE;
            .
            .
            .
END PROCESS1;

PROCESS2:   PROCEDURE;
            .
            .
            .
END PROCESS2;
```

Fig. 5-16. Creation of new processes in PL/I. The PL/I term for a process is a task.

programs are compiled has virtual instructions for creating new processes. In PL/I a process is called a **task**. Figure 5-16 shows how creation of a new process can be expressed in PL/I. The word "TASK" in the CALL statements informs the PL/I compiler that a new process is to be created for the procedure, to allow it to run in parallel with the main program. After the statement

```
CALL PROCESS2 TASK (T2);
```

has been executed, three processes will be running: the main one and the two newly created ones, named T1 and T2.

Some operating systems allow a process to maintain complete control over the processes it creates. To this end, virtual instructions exist for a process to stop, restart, examine, and destroy the processes it has created. Other operating systems give a process less control over the processes it creates: once a process has been created, there is no way for the creating process to forcibly stop, restart, examine, or destroy it.

5.3.2. Race Conditions

In this section the difficulties involved in synchronizing parallel processes will be explained by means of a detailed example. A solution to these difficulties will be given in the following section. Consider a system consisting of two independent processes, process 1 and process 2, which communicate via a shared buffer in main memory. For simplicity, we will call process 1 the producer and process 2 the consumer. The producer computes prime numbers and puts them into the buffer one at a time. The consumer removes them from the buffer one at a time and prints them.

These two processes run in parallel at different rates. If the producer discovers that the buffer is full, it halts (goes to sleep). Later, when the consumer has removed a number from the buffer, it sends a signal to the producer to "wake it up"—that is, restart it. Similarly, if the consumer discovers that the buffer is empty, it halts. When the producer has put a number into the empty buffer, it wakes up the sleeping consumer.

In this example we will use a circular buffer of the type used on the Cyber 70. The pointers IN and OUT will have their usual meanings: namely, IN points to the next free word, where the producer will put the next prime, and OUT points to the next number to be removed by the consumer. When IN = OUT, the buffer is empty. When there has been a sudden burst of input and IN has wrapped around and is only one behind OUT (e.g., IN = 52, OUT = 53), the buffer is full. The last word is not used, for if it were, there would be no way to tell if IN = OUT meant a full buffer or an empty one. Initially, IN = OUT = 1.

Figure 5-17 shows the programs used by the producer and the consumer. After the producer has found the next prime number in statement P1, it checks (in P2) to see if IN is one behind OUT. If it is (e.g., IN = 62 and OUT = 63), the buffer is full and the producer halts. If the buffer is not full, the new prime is inserted into the buffer (P3) and IN is incremented (P4). If the new value of IN is one ahead of OUT (P5) (e.g., IN = 17 and OUT = 16), IN and OUT must have been equal before IN

```
PRODUCER:   PROCEDURE;

     /* THE PRODUCER PUTS PRIME NUMBERS IN A SHARED BUFFER FOR SUBSEQUENT
        PRINTING. WHEN THE BUFFER IS FULL, THE PRODUCER GOES TO SLEEP.
        WHEN THE CONSUMER SENDS A WAKE UP SIGNAL, THE PRODUCER CONTINUES
        EXECUTING AT P3.  */

     DECLARE (IN,OUT,BUFFER(1:00))  FIXED BINARY EXTERNAL,
             PRIME FIXED BINARY INITIAL(1);

     DO WHILE (PRIME < 1000000);
        P1:  CALL COMPUTE_NEXT_PRIME(PRIME);
        P2:  IF (IN = OUT-1)|(IN = OUT+99) THEN CALL GO_TO_SLEEP;
        P3:  BUFFER(IN) = PRIME;
        P4:  IN = IN + 1;  IF IN = 101 THEN IN = 1;
        P5:  IF (IN = OUT+1) | (IN = OUT-99) THEN CALL WAKE_UP_CONSUMER;
     END;

END PRODUCER;

CONSUMER:  PROCEDURE;

     /* THE CONSUMER TAKES NUMBERS OUT OF THE BUFFER AND PRINTS THEM.
        IF THE BUFFER BECOMES EMPTY, THE CONSUMER GOES TO SLEEP. WHEN
        THE PRODUCER SENDS A WAKE UP SIGNAL, THE CONSUMER CONTINUES
        EXECUTING AT C2.*/

     DECLARE (IN,OUT,BUFFER(1:100)) FIXED BINARY EXTERNAL,
             EMIRP FIXED BINARY INITIAL(1);

     DO WHILE (EMIRP<1000000);
        C1:  IF IN = OUT THEN CALL GO_TO_SLEEP;
        C2:  EMIRP = BUFFER(OUT);
        C3:  OUT = OUT + 1;  IF OUT = 101 THEN OUT = 1;
        C4:  IF (OUT = IN + 2) | (OUT = IN - 98) THEN CALL WAKE_UP_PRODUCER;
        C5:  PUT SKIP LIST(EMIRP);
     END;
END CONSUMER;
```

Fig. 5-17. Parallel processing with a fatal race condition.

was incremented. The producer concludes that the buffer was empty and that the consumer was, and still is, halted. Therefore the producer sends a signal to wake up the consumer. Finally the producer begins looking for the next prime.

The consumer's program is structurally similar. First, a test is made (C1) to see if the buffer is empty. If it is, there is no work for the consumer to do, so it halts. If the buffer is not empty, it removes the next number to be printed (C2) and increments OUT (C3). If OUT is two positions ahead of IN at this point (C4), it must have been one ahead of IN before it was just incremented. Because IN = OUT − 1 is the "buffer full" condition, the producer must have been halted, and thus the consumer sends a signal to wake the producer up. Finally, the number is printed (C5) and the cycle repeats.

Unfortunately, this design has a fatal flaw, as shown in Fig. 5-18. Remember that the two processes run asynchronously and at different, possibly varying, speeds.

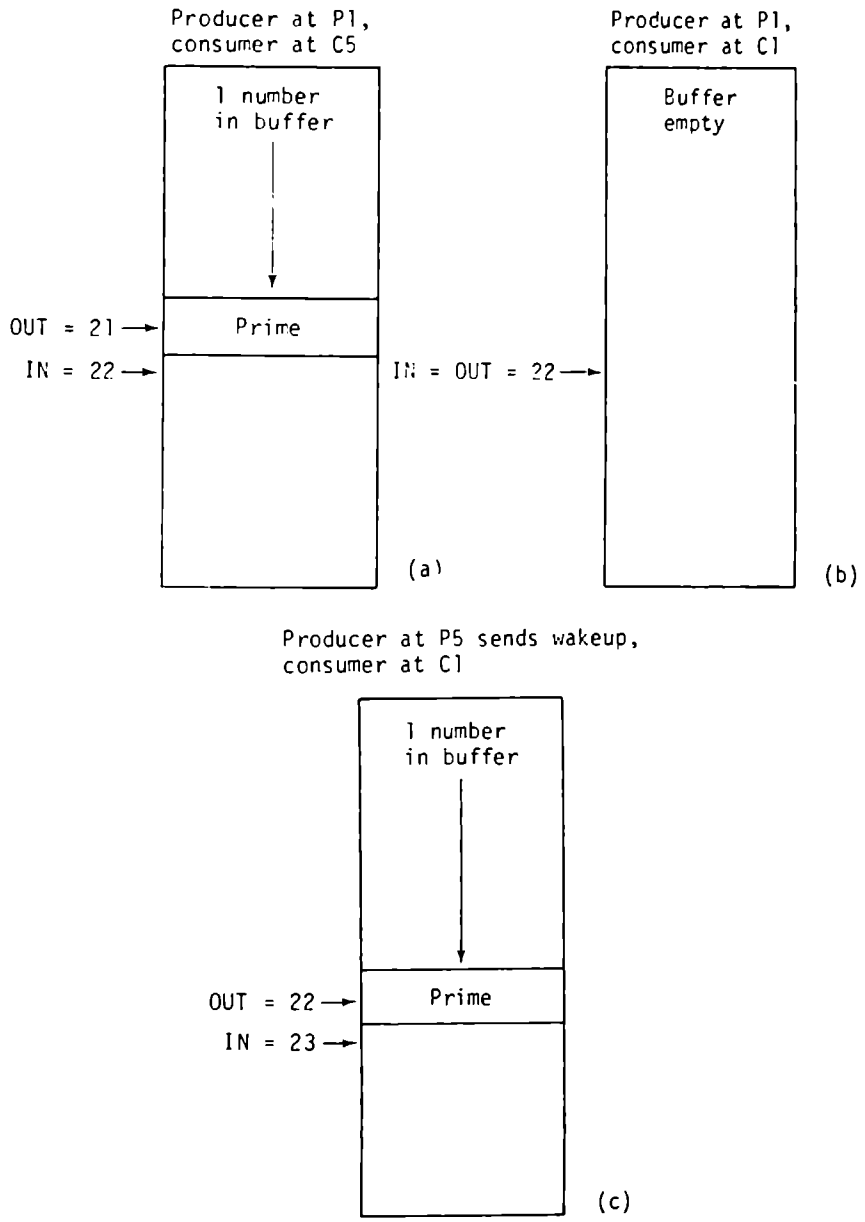


Fig. 5-18. Failure of the producer-consumer communication mechanism.

Consider the case where only one number is left in the buffer, in word 21, and IN = 22 and OUT = 21, as shown in Fig. 5-18(a). The producer is at statement P1 looking for a prime, and the consumer is busy at C5 printing out the number in position 20. The consumer finishes printing the number, makes the test at C1, and takes the last number out of the buffer at C2. It then increments OUT. At this instant, both IN

and OUT have the value 22. The consumer prints the number and then goes to C1 where it fetches IN and OUT from memory in order to compare them, as shown in Fig. 5-18(b).

At this very moment (after the consumer has fetched IN and OUT but before it has compared them), the producer has found the next prime. It puts the prime into the buffer at P3 and increments IN at P4. Now IN = 23 and OUT = 22. At P5 the producer discovers that IN = OUT + 1. It therefore (incorrectly) concludes that the consumer must be sleeping, so it sends a signal to wake it up, as shown in Fig. 5-18(c). Of course, the consumer is still awake; thus the wakeup signal is lost. The producer begins looking for the next prime.

At this point in time, the consumer continues. It has already fetched IN and OUT from memory before the producer put the last number in the buffer. Since they both have the value 22, the consumer halts. Now the producer finds another prime. It checks the pointers and finds IN = 24 and OUT = 22; therefore it assumes that there are two numbers in the buffer (correct) and that the consumer is awake (incorrect). The producer continues looping, and eventually it fills the buffer up and halts. Now both processes are halted and will remain halted forever.

The difficulty here is that between the time the consumer fetched IN and OUT and the time it actually halted, the producer snuck in, discovered that IN = OUT, assumed that the consumer was sleeping (which it was not yet), and sent a wakeup signal that was lost because the consumer was still awake. This is known as a **race condition**, since the method's success depends on who wins the race to test IN and OUT after OUT is incremented.

5.3.3. Process Synchronization Using Semaphores

There are at least two solutions to this race condition. One solution consists of equipping each process with a "wakeup waiting bit." Whenever a wakeup is sent to a process that is still running, its wakeup waiting bit is set. Whenever the process halts when the wakeup waiting bit is set, it is immediately restarted and the wakeup waiting bit is cleared. The wakeup waiting bit stores superfluous wakeup signals for future use.

Although this method solves the race condition when there are only two processes, it fails in the general case of N communicating processes, because as many as N − 1 wakeups may have to be saved. Of course, each process could be equipped with N − 1 wakeup waiting bits to allow it to count to N − 1 in the unary system, but this solution is rather clumsy.

Dijkstra (1968c) proposed a more general solution to the problem of synchronizing parallel processes. Somewhere in memory there are two nonnegative integer variables called **semaphores**. Two level 3 instructions that operate on semaphores, UP and DOWN, are provided by the operating system. UP adds 1 to a semaphore and DOWN subtracts 1 from a semaphore.

If a DOWN instruction is performed on a semaphore that is greater than 0, the semaphore is decremented by 1 and the process doing the DOWN continues. If, however, the semaphore is 0, the DOWN cannot complete; and the process doing the DOWN is halted and remains halted until the other process performs an UP on that semaphore.

The UP instruction checks to see if the semaphore is 0. If it is and the other process is halted on it, the semaphore is increased by 1. The halted process can then complete the DOWN operation that halted it, resetting the semaphore to 0 and allowing both processes to continue computing. An UP instruction on a nonzero semaphore simply increases it by 1. In essence, a semaphore provides a counter to store wakeups for future use, so that they will not be lost. An essential property of semaphore instructions is that once a process has initiated an instruction on a semaphore, no other process may access the semaphore until the first process has either completed its instruction or has been halted trying to perform a DOWN on a 0. Figure 5-19 summarizes the properties of the UP and DOWN instructions.

	Value of the semaphore before the instruction	
Instruction	Semaphore = 0	Semaphore > 0
UP	Semaphore = semaphore + 1 If the other process was halted attempting to complete a DOWN instruction on this semaphore, it may now complete the DOWN and continue running	Semaphore = semaphore + 1
DOWN	Process halts until the other process UP's this semaphore	Semaphore = semaphore - 1

Fig. 5-19. The effect of a semaphore instruction.

Figure 5-20 shows how the race condition can be eliminated through the use of semaphores. There are two semaphores, AVAILABLE, which is initially 100 (the buffer size), and FILLED, which is initially 0.

The producer starts executing at P1 in Fig. 5-20, and the consumer starts executing at C1. The Down instruction on FILLED halts the consumer processor immediately. When the producer has found the first prime, it executes a DOWN instruction on AVAILABLE, setting it to 99. At P5 it does an UP on FILLED, making it 1. This step releases the consumer, which is now able to complete its DOWN instruction. At this point, FILLED is 0 and both processes are running.

Let us now reexamine the race condition. At a certain point in time, IN = 22, OUT = 21, the producer is at P1, and the consumer is at C5. The consumer finishes what it was doing and gets to C1, where it DOWNs the semaphore, which had the

```
PRODUCER:   PROCEDURE;

    /* THIS IS AN IMPROVED VERSION OF THE PRODUCER PROCEDURE.  SEMAPHORES ARE
       USED TO PROVIDE THE SYNCHRONIZATION NECESSARY TO AVOID RACE CONDITIONS.
       THE PROCEDURES UP AND DOWN ARE MACHINE LANGUAGE PROCEDURES THAT EXECUTE
       THE UP AND DOWN INSTRUCTIONS, RESPECTIVELY.  IF THE LEVEL 3 MACHINE BEING
       USED DOES NOT HAVE THESE INSTRUCTIONS (OR THEIR EQUIVALENT), THIS
       SOLUTION TO THE PRODUCER - CONSUMER PROBLEM CANNOT BE USED.  AVAILABLE AND
       FILLED MUST BE INITIALIZED TO 100 AND 0 RESPECTIVELY, BY THE MAIN PROGRAM. */
    DECLARE (AVAILABLE, FILLED,IN,OUT,BUFFER(1:100)) FIXED BINARY EXTERNAL,
            PRIME FIXED BINARY INITIAL(1);

    DO WHILE (PRIME < 1000000);
       P1:  CALL COMPUTE_NEXT_PRIME(PRIME);
       P2:  CALL DOWN(AVAILABLE);
       P3:  BUFFER(IN) = PRIME;
       P4:  IN = IN + 1;  IF IN = 101 THEN IN = 1;
       P5:  CALL UP(FILLED);
    END;

END PRODUCER;

CONSUMER:    PROCEDURE;

    /* THIS IS THE IMPROVED VERSION OF THE CONSUMER, USING SEMAPHORES.*/

    DECLARE (AVAILABLE,FILLED,IN,OUT,BUFFER(1:100)) FIXED BINARY EXTERNAL,
            EMIRP FIXED BINARY INITIAL(1);

    DO WHILE (EMIRP < 1000000);
       C1:  CALL DOWN(FILLED);
       C2:  EMIRP = BUFFER(OUT);
       C3:  OUT = OUT + 1;  IF OUT = 101 THEN OUT = 1;
       C4:  CALL UP(AVAILABLE);
       C5:  PUT SKIP LIST(EMIRP);
    END;

END CONSUMER;
```

Fig. 5-20. Parallel processing using semaphores.

value 1 before the DOWN instruction and 0 after it. The consumer then takes the last number out of the buffer and UPs AVAILABLE, making it 100. The consumer prints the number and goes to C1. Just before the consumer can perform the DOWN instruction, the producer finds the next prime and in quick succession executes statements P2, P3, and P4.

At this point, FILLED is 0. The producer is about to UP it and the consumer is about to DOWN it. If the consumer executes its instruction first, it will be halted until the producer releases it (by performing the UP). On the other hand, if the producer executes its instruction first, the semaphore will be set to 1 and the consumer will not halt at all. In both cases, no wakeup is lost.

The essential property of the semaphore operations is that they are indivisible. Once a semaphore operation has been initiated, no other processor can use the

semaphore until the first processor has either completed the operation or been halted trying. Furthermore, with semaphores, no wakeups are lost. In contrast, the IF statements of Fig. 5-17 are not indivisible. Between the evaluation of the condition and the execution of the THEN or ELSE part, another process can send a wakeup signal.

In effect, the problem of process synchronization has been eliminated by declaring the UP and DOWN instructions to be indivisible. In order for these level 3 instructions to be indivisible, the operating system must prohibit two or more processes from using the same semaphore at the same time.

Synchronization using semaphores is a technique that works for arbitrarily many processes. Several processes may be halted, attempting to complete a DOWN instruction on the same semaphore. When some other process finally performs an UP on that semaphore, one of the waiting processes is allowed to complete its DOWN instruction and continue running. The semaphore value remains 0, and the other processes continue waiting.

An analogy may make the nature of semaphores clearer. Imagine a picnic with 20 volleyball teams divided into 10 games (processes), each playing on its own court, and a large basket (the semaphore) for the volleyballs. There are, unfortunately, only seven volleyballs at this picnic. At any instant, there are between zero and seven volleyballs in the basket (the semaphore has a value between 0 and 7). Putting a ball in the basket is an UP instruction, because it increases the value of the semaphore. Taking a ball out of the basket is a DOWN instruction, because it decreases the value of the semaphore.

At the start of the picnic, each court sends a player to the basket to get a volleyball. Seven of them successfully manage to get a volleyball (complete the DOWN instruction), but three are forced to wait for a volleyball, (i.e., fail to complete the DOWN instruction). Their games are halted temporarily. Eventually one of the other games finishes and one of its players puts the ball into the basket (executes an UP instruction). This operation allows one of the three players waiting around the basket to get a ball (complete an unfinished DOWN instruction), allowing his game to continue. The other two games remain halted until two more balls are put into the basket.

5.3.4. Instructions for Interprocess Communication

Although semaphores can be used to synchronize parallel processes, they do not provide a method for communicating much information from one process to another. In the example of the previous section, a shared buffer was used for transferring information. There are ways of handling interprocess communication and synchronization together.

One way is to associate a **mailbox** (or **event channel**) with each process. A level 3 instruction is provided for a process to SEND a message (consisting of an integer or a character string) to the mailbox of any other process. A level 3 instruction is also provided for a process to GET a message out of its mailbox, if one is waiting. If no message is available, the process is halted until some other process sends it a message.

The SEND message instruction is analogous to the UP instruction. The GET message instruction is analogous to the DOWN instruction. The mailbox is analogous to the semaphore, and the number of messages waiting in the mailbox is analogous to the semaphore value. The instructions GET and SEND message not only have the synchronization properties of the semaphores but they can convey information as well. It is the operating system's responsibility to implement these level 3 instructions, of course, including storing and retrieving messages when required.

5.4. OTHER LEVEL 3 INSTRUCTIONS

Just as each conventional machine level has its own unique instruction set, each operating system supports a different level 3 instruction set. A few other common level 3 instructions are discussed below.

5.4.1. Directory Management Instructions

In the early days of computing, people kept their programs and data on punched cards in their offices. As the programs and data grew in size and numbers, this situation became less and less desirable. Furthermore, users of a time-sharing system need to be able to examine and modify their programs from remote terminals. Together these factors led to the idea of using the computer's secondary memory (e.g., disk) as a storage place for programs and data as an alternative to people's offices. Information that is directly accessible to the computer without the need for human intervention is said to be on-line, as contrasted with off-line information, which requires human intervention (e.g., reading in a card deck) before the computer can access it.

On-line information is stored in files, making it accessible to programs via the file i/o instructions discussed in Sections 5.2.1 and 5.2.2. However, additional instructions are needed to keep track of the information stored on line and to protect it from unauthorized use.

Many operating systems make a distinction between scratch files, which a program creates and uses temporarily while it is executing, and permanent files, which are saved until explicitly deleted by their owner. For each user, the operating system maintains a **directory** or **catalog** of his permanent files being saved in secondary memory. Figure 5-21 shows a typical directory organization. Level 3 instructions are provided for at least the following functions:

1. Make a file permanent and enter it in the owner's directory.
2. Delete a permanent file from the directory.
3. Rename a permanent file.
4. Change the protection status of a permanent file.

Various protection schemes are in use. The simplest one is for the owner of each

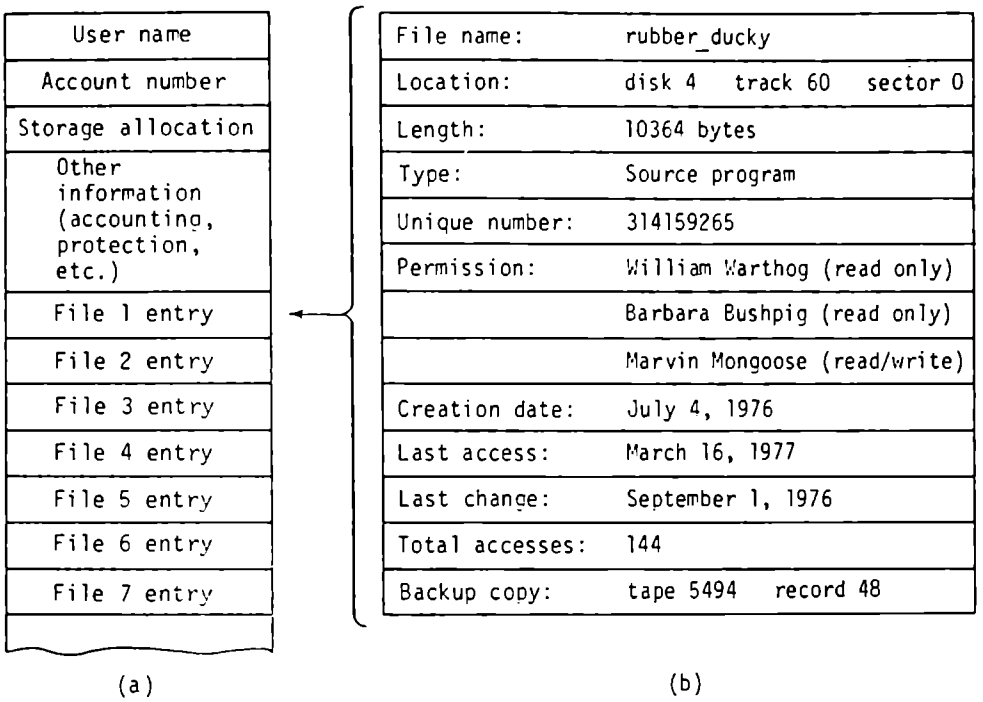


Fig. 5-21. (a) A user file directory. (b) The contents of a typical entry in a file directory.

file to specify a secret password for each permanent file. When attempting to access a permanent file, a program must supply the password, and the operating system then checks to see if it is correct before permitting the access. Another protection method is for the owner of each permanent file to provide an explicit list of people whose programs may access that file.

Some operating systems allow users to maintain more than one file directory. The MULTICS operating system for the Honeywell 6180, for example, allows each user to have as many directories as he needs. Each directory is itself a file and, as such, may be listed in an other directory, thus giving a tree of directories. Multiple directories are particularly useful for programmers working an several projects. They can then group all the files related to one project together in one directory; while working on that project, they will not be distracted by unrelated files.

5.4.2. Reconfiguring the Level 3 Machine

Every computer at every level has certain properties and features that are considered part of the virtual "hardware" configuration. For example, a level 2 machine might have 32,768 words of memory, one disk, and one tape drive. There is nothing a level 2 program can do to either increase or decrease the amount of memory or

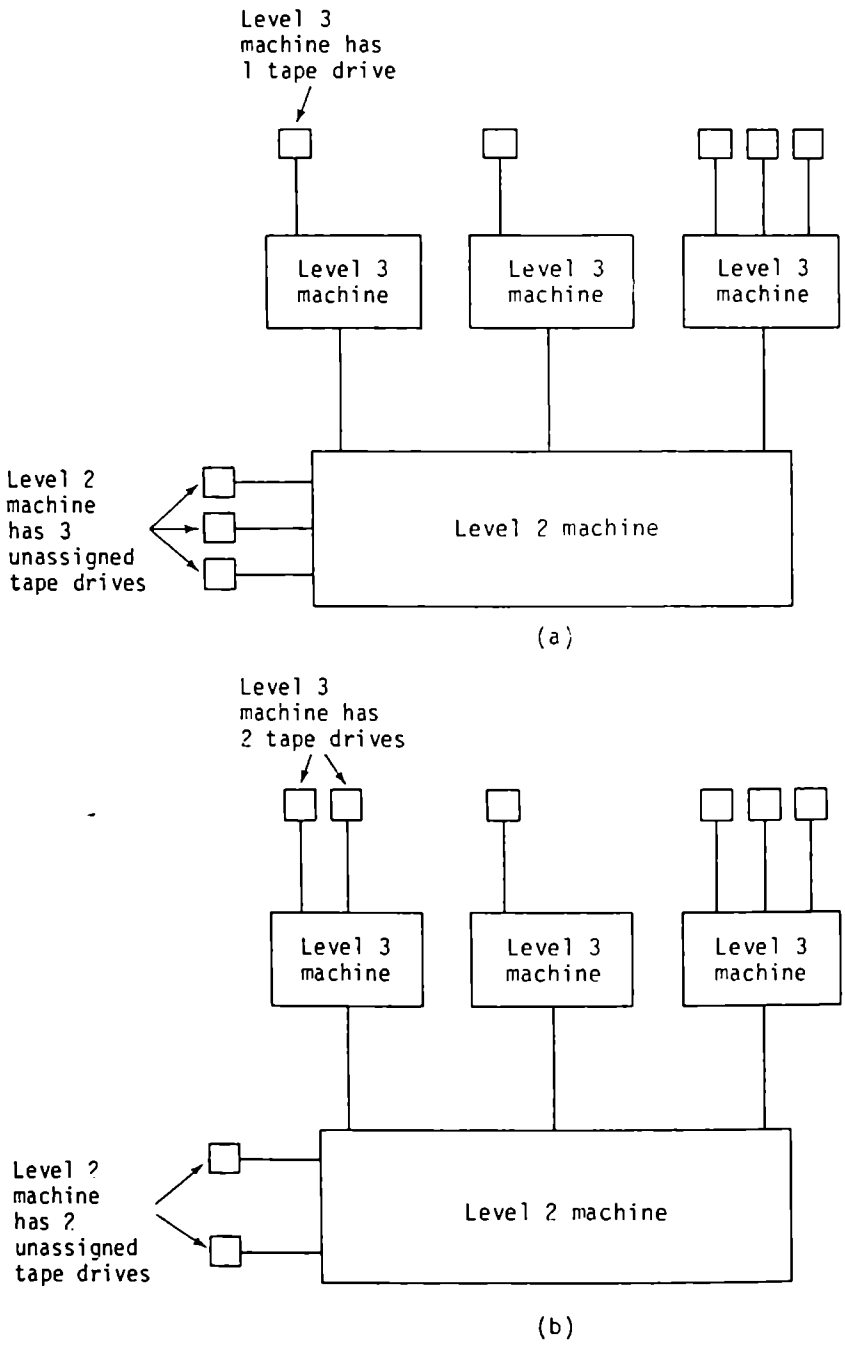


Fig. 5-22. A level 2 machine supporting three level 3 machines. The computer has a total of eight hardware tape drives. (a) Tape drive assignment at a certain point in time. (b) Tape drive assignment after one of the level 3 machines has requested another tape drive.

the number of disk drives or tape drives available. The computer center could buy more tape drives, but doing that is quite different from having a machine on which the number of tape drives could be affected by the program. A program that chooses not to use the available tape drives in no sense decreases the number of tape drives available. It merely does not use them.

Level 3 machines frequently have an instruction to alter the configuration of the virtual machine. For example, there is often an instruction to add another tape drive or to add some more memory. Sometimes a program can execute an instruction that attaches an i/o device previously not available, such as a plotter or a microfilm camera and visual display.

How is this possible? In order for a level 3 machine to have attached to it an i/o device that was not there previously, the i/o device must have been attached to the level 2 machine supporting it. The level 2 machine is under no obligation to provide all its resources to the level 3 machine it is supporting, especially if it is supporting several at once, as is the case in a multiprogramming or time-sharing system.

If a level 3 machine requests some device and the level 2 machine has one available, the latter can assign it to the level 3 machine, which can then use the device as though it were always there. When level 3 machine is finished using the device, it can issue a virtual instruction detaching the device. Detached devices would return to level 2 for future connection to another virtual machine. Figure 5-22 illustrates a group of level 3 machines being supported in parallel by the multiprogramming of a single level 2 machine. The figure shows the effect of requesting a tape drive.

What happens if a level 3 machine issues an instruction to attach a tape drive, and all the tape drives are already attached to other level 3 machines being supported by the level 2 machine? The level 3 machine will be stopped and will not be able to continue until the level 2 machine has carried out the ATTACH instruction. When another level 3 machine releases a tape drive, it can be attached to the requesting level 3 machine, which can then be restarted.

An interesting situation can arise if a level 2 machine with two or more of some resource, such as a tape drive, is supporting two or more level 3 machines. Suppose that each of two level 3 machines, V1 and V2, requests one of the two available tape drives. The requests are granted and both continue computing. Then V1 executes an instruction requesting the other tape drive. Its progress is stopped by the operating system, and it must wait until V2 releases its tape drive. But instead of releasing its tape drive, V2 also requests another tape drive, so it, too, is stopped. Neither V1 nor V2 can continue until a tape drive becomes available; and since both are stopped, neither one will ever release the one tape drive it has. This situation is called a **deadlock** or **deadly embrace**. Readers interested in solutions to this difficulty are referred to the suggestions for further reading, in Chapter 8.

5.5. VIRTUAL MEMORY

In the early days of computers, memories were expensive and small. The IBM 650, the leading scientific computer of its day (late 1950s), had only 2000 words of

memory. One of the first ALGOL 60 compilers was written for a computer with barely 1000 words of memory. In those days the programmer spent a large portion of his time trying to squeeze his program into the tiny memory. Often it was necessary to use an algorithm that ran a great deal slower than another, better algorithm simply because the better algorithm was too big—that is, a program using the better algorithm could not be fitted into the computer's memory.

The traditional solution to this problem has been the use of secondary memory, such as tape, drum, and disk. The programmer divided his program up into a number of pieces, called overlays, each of which could fit in the memory. To run the program, the first overlay was brought in and it ran for a while. When it finished, it read in the next overlay and called it, and so on. The programmer himself was responsible for breaking the program into overlays, deciding where in the secondary memory each overlay was to be kept, arranging for the transport of overlays between main memory and secondary memory, and, in general, managing the whole overlay process without any help from the computer.

Although widely used for many years, this technique involved much work in connection with the overlay management. In 1961 a group of people at Manchester, England, proposed a method for performing the overlay process automatically, without the programmer even knowing that it was happening (Fotheringham, 1961). This method, now called **virtual memory**, had the obvious advantage of freeing the programmer from a great deal of annoying bookkeeping. It was used on a number of computers during the 1960s, mostly associated with research projects in computer systems design. By the early 1970s virtual memory had become available on most computers, including the 370 and PDP-11. The Cyber 70 does not have virtual memory.

5.5.1. Paging

The idea put forth by the Manchester group was to separate the concepts of address space and memory locations. Consider an example of a computer with a 16-bit address field in its instructions, and 4096 words of memory. A program on this computer can address 65,536 words of memory. The reason is that 65,536 16-bit numbers (addresses) exist. The number of addressable words depends only on the number of bits in an address and is in no way related to the number of memory words actually available. The **address space** for this computer consists of the numbers 0, 1, 2, . . . , 65535, since that is the set of possible addresses.

Before virtual memory was invented, people would have made a distinction between the addresses below 4096 and those equal to or above 4096. Although rarely stated in so many words, these two parts were regarded as the useful address space and the useless address space, respectively (the addresses above 4095 being useless because they did not correspond to actual memory addresses). People made little distinction between address space and actual memory addresses, because the hardware enforced a one-to-one correspondence between them.

The idea of separating the address space and the memory addresses is as follows. At any instant of time, 4096 words of memory can be directly accessed, but they need not correspond to addresses 0 to 4095. We could, for example, "tell" the computer that henceforth whenever address 4096 is referenced, memory word 0 is to be used. Whenever address 4097 is referenced, memory word 1 is to be used; and whenever address 8191 is referenced, memory word 4095 is to be used, and so forth. In other words, we have defined a mapping from the address space onto the actual memory addresses, as shown in Fig. 5-23.

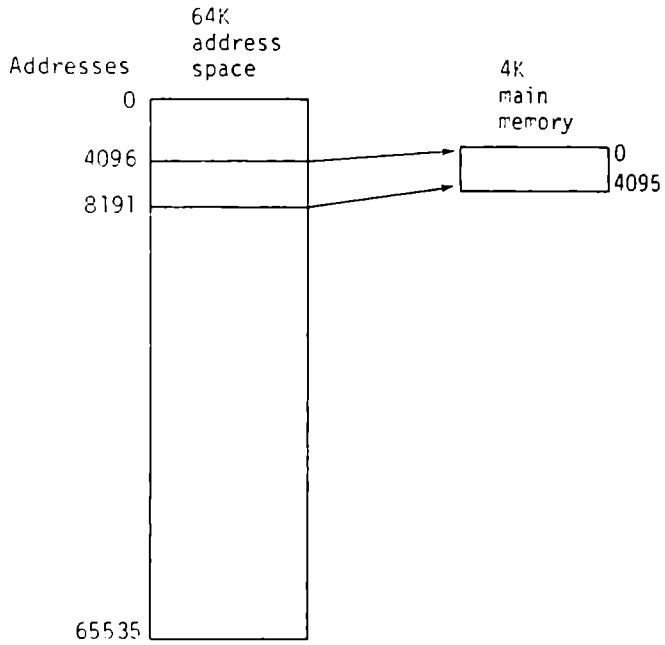


Fig. 5-23. A mapping in which addresses 4096–8191 are mapped onto main memory addresses 0–4096.

In terms of this picture of mapping addresses from the address space onto the actual memory locations, a 4K machine without virtual memory simply has a fixed mapping between the addresses 0 to 4095 and the 4096 words of memory. An interesting question is, "What happens if a program jumps to an address between 8192 and 12,287?" On a machine without virtual memory, the program would cause an error trap that would print a suitably rude message, such as "Nonexistent memory referenced," and terminate the program. On a machine with virtual memory, the following sequence of steps would occur:

1. The contents of main memory would be saved in the secondary memory.
2. Words 8192 to 12,287 would be located in the secondary memory.
3. Words 8192 to 12,287 would be loaded into main memory.

4. The address map would be changed to map addresses 8192 to 12,287 onto memory locations 0 to 4095.
5. Execution would continue as though nothing unusual had happened.

This technique for automatic overlaying is called **paging**, and the chunks of program read in from secondary memory are called **pages**.

A more sophisticated way of mapping addresses from the address space onto the actual memory addresses is also possible. For emphasis, we will call the addresses that the program can refer to the **virtual address space** and the actual, hard-wired memory addresses the **physical address space**. A memory map exists that relates virtual addresses to physical addresses. It is presumed that there is enough room in the secondary memory (e.g., a drum or a disk) to store the whole program and its data.

Programs are written just as though there were enough main memory for the whole virtual address space, even though such is not the case. Programs may load from or store into any word in the virtual address space, or jump to any instruction located anywhere within the virtual address space, without regard to the fact that actually there is not enough physical memory. In fact, the programmer can write his programs without even being aware that virtual memory exists. He simply thinks that the computer has a big memory.

This point is crucial and will be contrasted later with segmentation, where the programmer must be aware of the existence of segments. To emphasize it once more, paging gives the programmer the illusion that a large, continuous, linear main memory, the same size as the address space, exists when, in fact, the main memory available may be smaller than the address space. The simulation of this large main memory by paging cannot be detected by the program (except by running timing tests); whenever an address is referenced, the proper instruction or data word appears to be present. Because the programmer can program as though paging did not exist, the paging mechanism is said to be **transparent**.

The idea that a programmer may use some nonexistent feature without being concerned with how it works is not new to us, after all. The instruction set of a level 2 computer is nonexistent in the sense that none of the instructions is a hardware primitive but is, in fact, carried out by software at level 1. Similarly, the level 3 programmer can use the virtual memory without worrying about how it works. The level 2 programmers who write the operating system must, of course, know exactly how it works.

5.5.2. Implementation of Paging

One essential requirement for a virtual memory is a secondary memory in which to keep the complete program. It is conceptually simpler if one thinks of the copy of the program in the secondary memory as the original one and the pieces brought into main memory every now and then as copies, rather than the other way around.

Naturally, it is important to keep the orginal up to date. When changes are made to the copy in main memory, they should be reflected in the original (eventually), too.

The virtual address space is broken up into a number of equally sized pages. Page sizes ranging from 512 to 4096 addresses per page are common at present. The page size is always a power of 2. The physical address space is broken up into pieces in a similar way, each piece being the same size as a page, so that each piece of main memory is capable of holding exactly one page. These pieces of main memory into which the pages go are called **page frames**. In Fig. 5-23 the main memory only contains one page frame. In practical designs, it will contain anywhere from 8 to 512 or even more in a very large machine. Figure 5-24 illustrates a possible way to divide up a 64K address space.

The virtual memory of Fig. 5-24 would be implemented at level 2 by means of

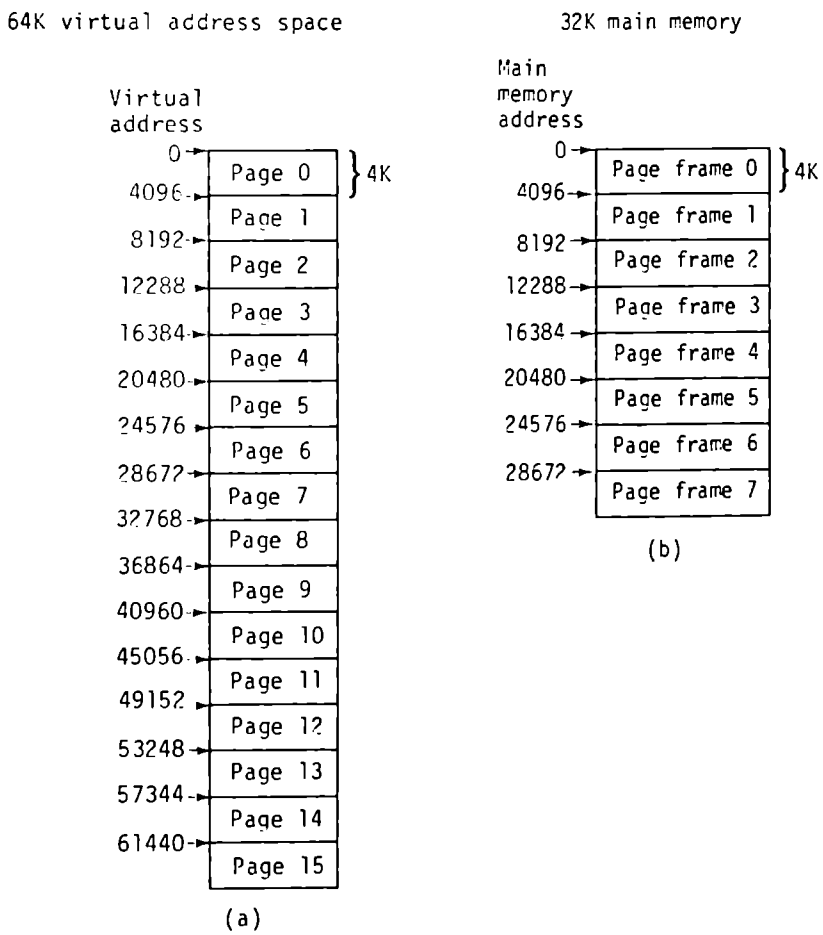


Fig. 5-24. (a) A 64K address space divided up into 16 pages of 4K each. (b) A 32K main memory divided up into eight page frames of 4K each.

a 16-word table called the **page table**. When the program tried to reference its memory, whether to fetch data, store data, fetch instructions, or jump, it would first generate a 16-bit address, corresponding to a virtual address between 0 and 65,535. Indexing, indirect addressing, and all the usual techniques may be used to generate this address.

In this example, the 16-bit address is taken as a 4-bit virtual page number and a 12-bit address within the selected page, as shown in Fig. 5-25. In this figure the

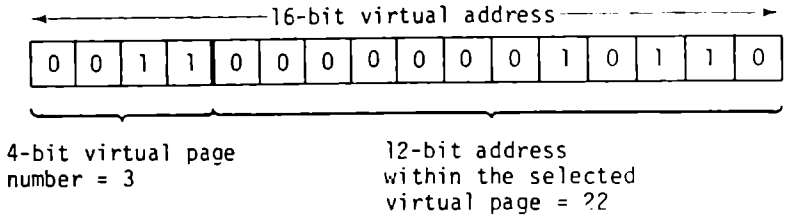


Fig. 5-25. The virtual addresses of Fig. 5-24 consist of a 4-bit virtual page number and a 12-bit offset within the selected page.

16-bit address is 12,310, which is regarded as address 22 of page 3. The relation between pages and virtual addresses for this example is shown in tabular form.

Page	Virtual Addresses
0	0–4095
1	4096–8191
2	8192–12287
3	12288–16383
4	16384–20479
5	20480–24575
6	24576–28671
7	28672–32767
8	32768–36863
9	36864–40959
10	40960–45055
11	45056–49151
12	49152–53247
13	53248–57343
14	57344–61439
15	61440–65535

If address 0 of page 3 is 12,288, address 22 must be 12,310.

Having discovered that virtual page 3 is needed, the operating system must find out where virtual page 3 is located. There are nine possibilities: eight page frames in main memory, or somewhere in secondary memory, since not all the virtual pages can be in main memory at once. To find out which of these nine possibilities is true, the operating system looks in the page table, which has one entry for each of the 16 virtual pages.

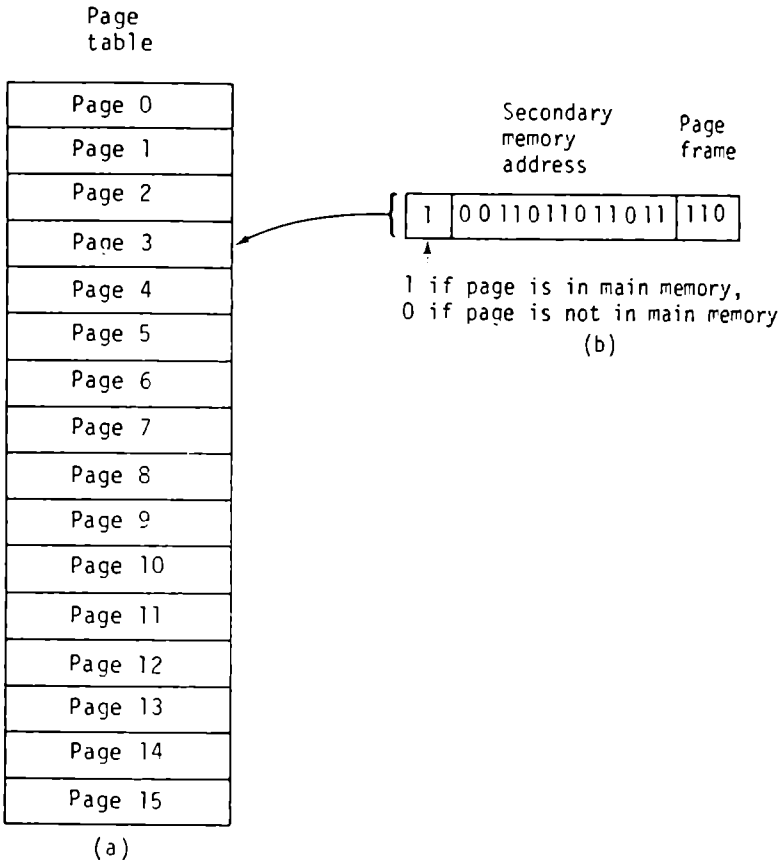


Fig. 5-26. (a) The page table must contain as many entries as there are virtual pages. (b) The example page table has three fields per entry.

The example page table of Fig. 5-26 has three fields. The first is a bit that is 0 if the page is not in main memory and 1 if it is. The second gives the address where the virtual page is kept in secondary memory (e.g., a drum track and sector) when not in main memory. This address is needed so that the page can be found and brought in when necessary and later returned to its original place in secondary memory when no longer needed in main memory. The third is a three-bit field giving the page frame where the page is located if it is in main memory. If the page is not in main memory, field 3 is ignored.

Assuming that the virtual page is in main memory, the three-bit page frame specifies where the page is. The page frame number is then gated into the leftmost three bits of the MAR, and the address within the virtual page—that is, the rightmost 12 bits of the original address—are gated into the rightmost 12 bits of the MAR. In this way, a main memory address can be formed, as shown in Fig. 5-27. The three-bit page frame plus the 12-bit offset give a 15-bit address, which is precisely what is needed for the 32K main memory of Fig. 5-24, which we are considering. The hard-

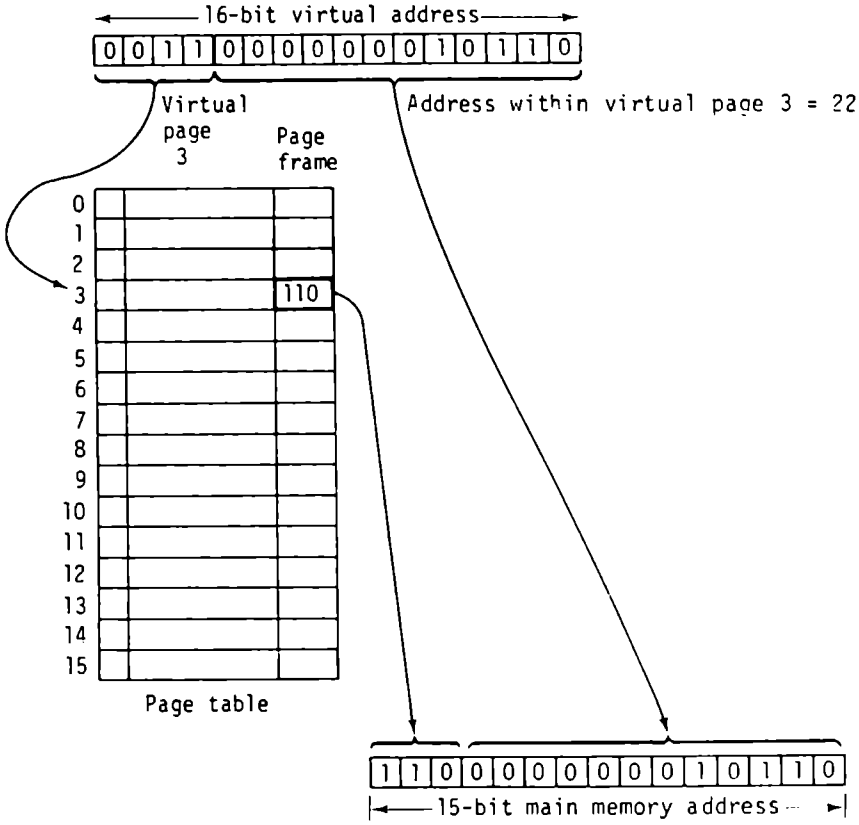


Fig. 5-27. Formation of a main memory address from a virtual address for a paged memory.

ware can now use this address and fetch the desired word into the MBR, or it can store the MBR into the desired word.

Figure 5-28 shows a possible mapping between virtual pages and physical pages. Virtual page 0 is in page frame 1. Virtual page 1 is in page frame 0. Virtual page 2 is not in main memory. Virtual page 3 is in page frame 2. Virtual page 4 is not in main memory, and so forth.

5.5.3. Demand Paging and the Working Set Model

In the preceding discussion it was assumed that the virtual page referenced was in main memory. However, that assumption will not always be true because there is not enough room in main memory for all the virtual pages. When a reference is made to an address on a page not present in main memory, it is called a **page fault**. After a page fault has occurred, it is necessary for the operating system to read in the required page from the secondary memory, enter its new physical memory location in the page table, and then repeat the instruction.

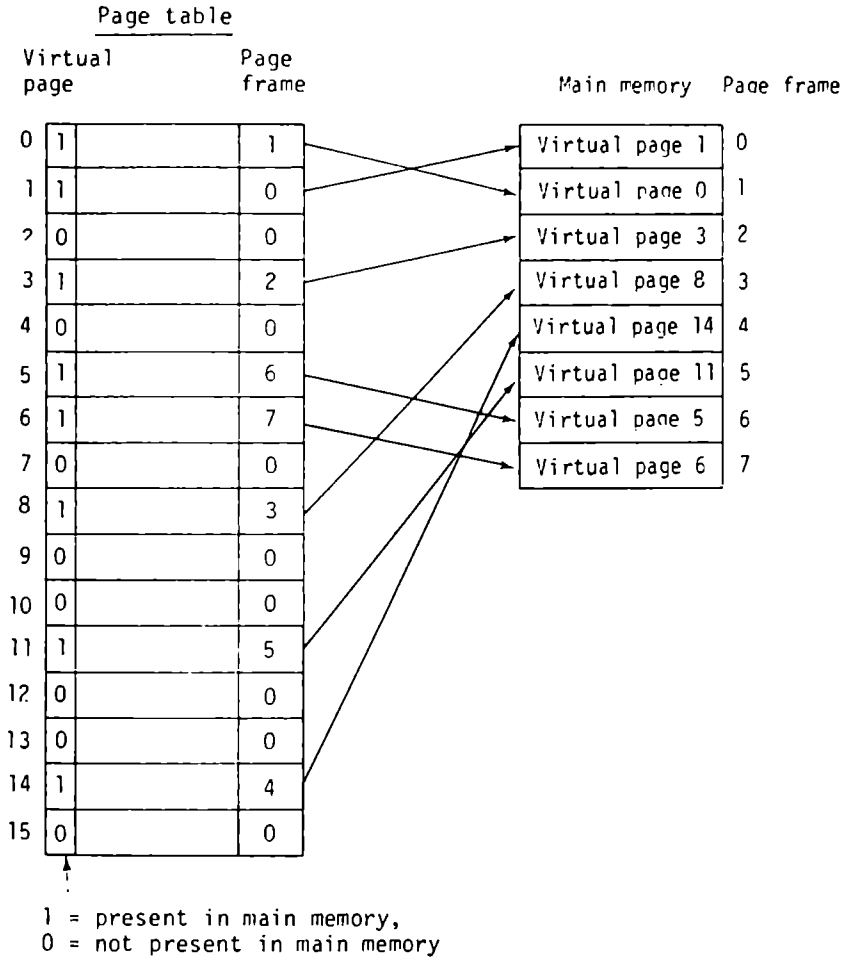


Fig. 5-28. A possible mapping of an address space with 16 pages onto a main memory with eight page frames.

It is possible to start a program running on a machine with virtual memory even though none of the program is in main memory. The page table merely has to be set to indicate that each and every virtual page is in the secondary memory, and not in main memory. When the CPU tries to fetch the first instruction, it immediately gets a page fault, which causes the page containing the first instruction to be loaded and entered in the page table. Then the first instruction can begin. If the first instruction has two addresses, with the two addresses on different pages, both different from the instruction page, two more page faults will occur, and two more pages will be brought in before the instruction can finally execute. The next instruction may possibly cause some more page faults, and so forth.

This method of operating a virtual memory system is called **demand paging**, in analogy to the well-known demand feeding algorithm for babies: when the baby

cries, you feed it (as opposed to feeding it at regular times of day). In demand paging, pages are brought in only when an actual request for a page occurs, not in advance.

The question of whether demand paging should be used or not is only relevant when a program first starts up. Once it has been running for a while, the needed pages will already have been collected in main memory. If the computer is time shared, and users are swapped out after running 100 msec or so, then each program will be restarted many times during the course of its run. Thus the question repeatedly becomes a critical one.

The alternative approach is based on the observation that most programs do not reference their address space uniformly but that the references tend to cluster on a small number of pages. A memory reference may fetch an instruction, it may fetch data, or it may store data. At any instant in time, t, there exists a set consisting of all the pages used by the k most recent memory references. Denning (1968) has called it the **working set**, $w(k, t)$. Since the $k + 1$ most recent references must have used all the pages used by the k most recent references, and possibly others, $w(k, t)$ is a monotonically nondecreasing function of k. The limit of $w(k, t)$ as k becomes very large is finite, since the program cannot reference more pages than its address space contains, and few programs will use every single page.

Figure 5-29 depicts the size of the working set as a function of k. The fact that most programs randomly access a small number of pages, but that this set changes slowly in time, explains the initial rapid rise of the curve and then the very slow rise for large k. For example, a program that is executing a loop occupying two pages, and using data occupying four pages, may reference all six pages every 1000 instructions, but the most recent reference to some other page may be a million references earlier, during the initialization phase. Because of this asymptotic behavior, the contents of the working set is not very sensitive to the value of k chosen, or, to state it another way, there exists a wide range of k values for which the working set is unchanged.

Since the working set varies slowly with time, it is possible to make a reasonable guess as to which pages will be needed when the program is restarted, on the basis

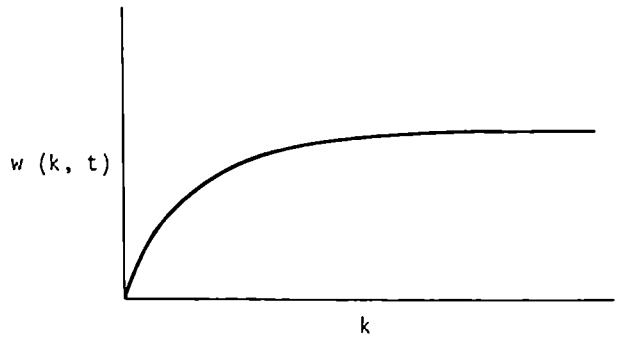


Fig. 5-29. The working set is the set of pages used by the k most recent memory references. The function $w(k, t)$ is the size of the working set at time t.

of its working set when it was last stopped. These pages could be then loaded in advance before starting the program up.

The argument in favor of bringing the working set into main memory in advance is that it can be brought in while some other program is running. When a program is started, it will not immediately generate a large number of time-consuming page faults, an event that may cause the CPU to be idle while the needed pages are being brought in. Remember that the time needed to read in a page from a drum is typically 20,000 instruction times or more.

The argument against bringing the working set into main memory in advance is that if the program is in transition between one working set and another, and has not yet settled down, a lot of work will have been done bringing in pages that are not going to be referenced. With demand paging, that situation cannot occur. The only pages that are loaded are those that are actually needed. The relative merits of these two strategies are still being debated.

5.5.4. Page Replacement Policy

It has tacitly been assumed that there is a vacant page frame in which to put the newly loaded page. In general, such will not be true, and it will be necessary to remove some page (i.e., copy it back to the secondary memory) in order to make room. Thus an algorithm that decides which page to remove is needed.

Choosing a page to remove at random certainly is not a good idea. If the page containing the instruction should happen to be the one picked, another page fault will occur as soon as an attempt is made to fetch the next instruction. Most operating systems try to predict which of the pages in memory is the least useful in the sense that its absence would have the smallest adverse effect on the running program. One way of doing so is to make a prediction when the next reference to each page will occur and remove the page whose next reference lies farthest in the future. In other words, rather than evict a page that will be needed shortly, try to select one that will not be needed for a long time.

One popular algorithm evicts the page least recently used because that page has a high a priori probability of not being in the current working set. It is called the **Least Recently Used** or LRU algorithm. Although it usually performs well, there are pathological situations, such as the one described below, where LRU fails miserably.

Imagine a program executing a large loop that extends over nine virtual pages on a machine with room for only eight pages in memory. After the program gets to page 7, the main memory will be as shown in Fig. 5-30(a). An attempt is eventually made to fetch an instruction from virtual page 8, which causes a page fault. A decision has to be made about which page to evict. The LRU algorithm will choose virtual page 0, since it has been used least recently. Virtual page 0 is removed and virtual page 8 is brought in to replace it, giving the situation in Fig. 5-30(b).

After executing the instructions on virtual page 8, the program jumps back to the top of the loop, to virtual page 0. This step causes another page fault. Virtual page 0, which was just thrown out, has to be brought back in. The LRU algorithm chooses

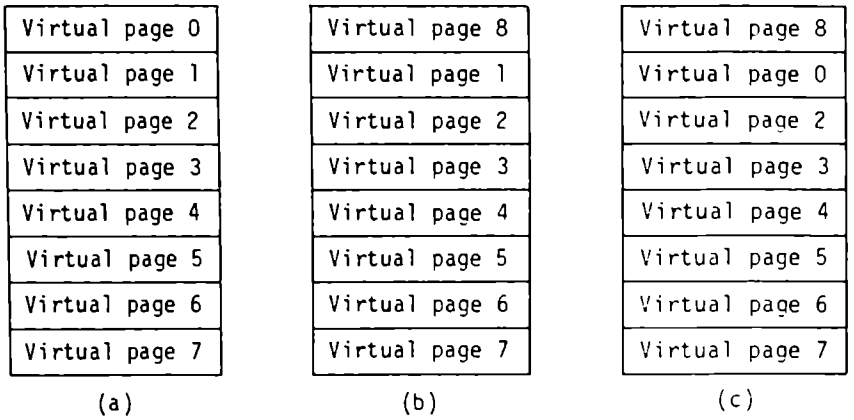


Fig. 5-30. Failure of the LRU page replacement algorithm for a nine page working set and an eight page frame memory. When the pages are used sequentially, LRU consistently makes the worst possible choice.

page 1 to be thrown out, producing the situation in Fig. 5-30(c). The program continues on page 0 for a little while and then tries to fetch an instruction from virtual page 1, causing a page fault. Page 1 has to be brought back in again, and page 2 will be thrown out.

It should be apparent by now that here the LRU algorithm is consistently making the worst choice every time. If, however, the available main memory exceeds the size of the working set, LRU tends to minimize the number of page faults.

Another algorithm is **First-In, First-Out**, or FIFO. FIFO removes the least recently loaded page, independent of when this page was last referenced. Associated with each virtual page is a counter, possibly kept in the page table. Initially all the counters are set to zero. After each page fault has been handled, the counter for each page presently in memory is increased by one, and the counter for the page just brought in is set to zero. When it becomes necessary to choose a page to remove, the page whose counter is highest is chosen. The fact that its counter is the highest means that it has witnessed the largest number of page faults. This means that it was loaded previous to the loading of any of the other pages in memory and therefore (hopefully) has a large a priori chance of no longer being needed.

If the working set is larger than the number of available page frames, no algorithm that is not an oracle will give good results, and page faults will be frequent. A program that generates page fault frequently and continuously is said to be **thrashing**. Needless to say, thrashing is an undesirable characteristic. If a program uses a large amount of virtual address space but has a small, slowly changing working set that fits in available main memory, it will give little trouble, even if, over its lifetime, it uses hundreds of times as many words of virtual memory as the machine has words of main memory.

5.5.5. The Dirty Bit

If a page about to be evicted has not been modified since it was read in (a likely occurrence if the page contains program rather than data), then it is not necessary to write it back into secondary memory, because an accurate copy already exists there. If it has been modified since it was read in, the copy in the secondary memory is no longer accurate, and the page must be rewritten.

If a way existed to tell whether a page has remained unchanged since it was read in (page is clean) or whether it, in fact, has been stored into (page is dirty), then all the rewriting of clean pages could be avoided, thus saving a lot of time. Some computers have one bit per page, in the page table, which is set to zero when a page is loaded and set to one by the microprogram or hardware whenever it is stored into. By examining this bit, the operating system can find out if the page is clean or dirty, and hence whether it need be rewritten or not. Such a bit is sometimes referred to as the **dirty bit**.

It is obviously desirable to maintain a high ratio of clean pages to dirty pages in order to minimize the chance that a rewrite will be required at the next page fault. On nearly all computers, pages can be copied from main memory to secondary memory, particularly drums and disks, while the CPU is computing. The CPU need only start the transfer and then it can return to its computation. If the secondary memory is a drum, and no transfers are waiting to be done, the CPU could choose a dirty page, preferably one likely to be written out shortly (due to its age), and issue a drum command to copy it to the drum. Copying it to the drum does not alter or destroy the copy in main memory, of course.

If the page were made dirty again immediately after the copying process, or even during it, the copying would have been done in vain, but since the drum was idle and the CPU was free to compute again as soon as it issued the drum command, the cost of performing the copy was not high. Writes to secondary memory made with the intention of making dirty pages clean have been called **sneaky writes**. It is still an open issue whether the amount of time saved swapping is enough to counterbalance the administration involved in setting up the sneaky writes.

5.5.6. The Hardware Map

If the operating system had to convert every level 3 machine instruction's virtual address into an actual address, a level 3 machine with virtual memory would run many times slower than one without virtual memory, and the whole idea would be impractical. To speed up the virtual-to-physical-address translation, the page table is usually maintained in special hardware registers, and the transformation from virtual address to actual address is done directly in hardware. Another way of doing it is to maintain the map in fast registers and let the microprogram do the transformation by explicit programming. Depending on the architecture of the microprogramming level, having the microprogram perform the transformation might be

almost as fast as doing it directly in the hardware, and it would require no special circuits or hardware modifications.

5.5.7. Page Size and Fragmentation

If the user's program and data accidently happen to fill an integral number of pages exactly, there will be no wasted space when they are in memory. If, on the other hand, they do not fill an integral number of pages exactly, there will be some unused space on the last page. For example, if the program and data need 26,000 words on a machine with 4096 words per page, the first six pages will be full, totaling $6 \times 4096 = 24576$ words, and the last page will contain $26000 - 24576 = 1424$ words. Since there is room for 4096 words per page, 2672 words will be wasted. Whenever the seventh page is present in memory, those words will take up precious main memory but will serve no useful function. The problem of these wasted words is called **fragmentation.**

If the page size is N words, the average amount of space wasted in the last page of a program by fragmentation will be N/2 words—a situation that suggests a small page size in order to minimize waste. On the other hand, a small page size means many pages, as well as a large page table. If the page table is maintained in hardware, a large page table means that more registers are needed to store it, a fact that increases the cost of the computer. In addition, more time will be required to load and save these registers whenever a process is started or stopped.

Furthermore, small pages make inefficient use of secondary memories with long rotational delays, such as drums and disks. Given that one is going to wait 10 msec or more before the transfer can begin, one would like to transfer a large block of information, since the transfer time is usually shorter than the rotational delay time. Generally it costs little extra time to read 1024 words as compared to 64 words. If a secondary memory with no rotational delay is being used, such as low-speed core or solid-state memory, then the total transfer time is proportional to the block size.

Small pages do have the advantage that if the working set consists of a large number of small, separated regions in the virtual address space, there may be less thrashing with a small page size than with a big one. For example, consider a program that is randomly accessing 20 widely separated regions of 100 words each. If a 1000 by 20 matrix A is stored with $A(1, 1)$, $A(2, 1)$, $A(3, 1)$, etc. in consecutive words, $A(1, 1)$, $A(1, 2)$ and $A(1, 3)$ will be 1000 words apart. A program performing some calculation on all the elements of the first 20 rows would use 20 regions of 100 words with 900 words separating these regions. If the page size were 2048 words, at least 10 pages, totaling 20,480 words, would be in the working set. If the page size were 128 words, then even if each region occupied parts of two pages, only 40 pages, totaling 5120 words, would be needed to run the program. If the main memory available was more than 5120 words but less than 20,480 words, the large page size would prohibit the complete working set from being in main memory, thereby causing thrashing, whereas the small page size would cause no problems.

5.5.8. Cache Memory

The principle behind a paged virtual memory can be generalized to a hierarchical memory. Some computers (including models of the 370 series) have a small, very high-speed memory called a **cache** memory (or simply a cache) in addition to the usual main memory and secondary memory. The cache holds those parts of the program and data that are most heavily used, the main memory holds those parts that are less heavily used, and the secondary memory holds those parts that are least heavily used.

The cache works as follows. Main memory is divided up into blocks. These blocks may be the same size as, or smaller than, the page size. Copies of the most heavily used blocks are kept in the cache, as well as in main memory. When a program tries to reference its address space, it presents a virtual address to the hardware (or microprogram). If the page containing that address is in main memory, the virtual address is translated into a main memory address; otherwise a page fault occurs and the page is brought into main memory.

The cache hardware then checks to see if the block referenced happens to be in the cache memory. If it is there, the contents of the addressed location are fetched from the cache, or the result is stored into the cache. If it is not there, a cache block fault, analogous to a page fault, occurs and the needed block is copied from main memory into the cache. After the block has been loaded into the cache, the instruction is executed again.

This is essentially a two-level paging system, with the cache treating physical main memory addresses as virtual addresses and cache addresses as real addresses. However, there are two significant differences between paging from secondary memory into main memory and paging from main memory into the cache:

1. Main memory has no rotational delay, as a drum does, and so the block can be copied from main memory to the cache immediately. Because the transfer will be completed within microseconds instead of milliseconds, the CPU can afford to stop and wait for it. When paging from a drum, the delay is so long that the CPU will usually find another process to run while waiting for the page.
2. Due to the very short time involved and the simplicity of the transfer, the cache-main memory paging system is handled completely by the microprogram without any assistance from the operating system.

The comments made earlier about replacement algorithms, dirty bits, and fragmentation apply equally well to the cache-main memory system as to the main memory-secondary memory system to which they were originally applied. Furthermore, there is no reason why a memory hierarchy must be limited to only three levels. Figure 5-31 illustrates a six-level memory hierarchy. There is a steady decrease in speed and an increase in storage capacity from the cache to the top level. The access-time ratio between the top and the bottom can be a factor of 10^9 (e.g., 50 nsec

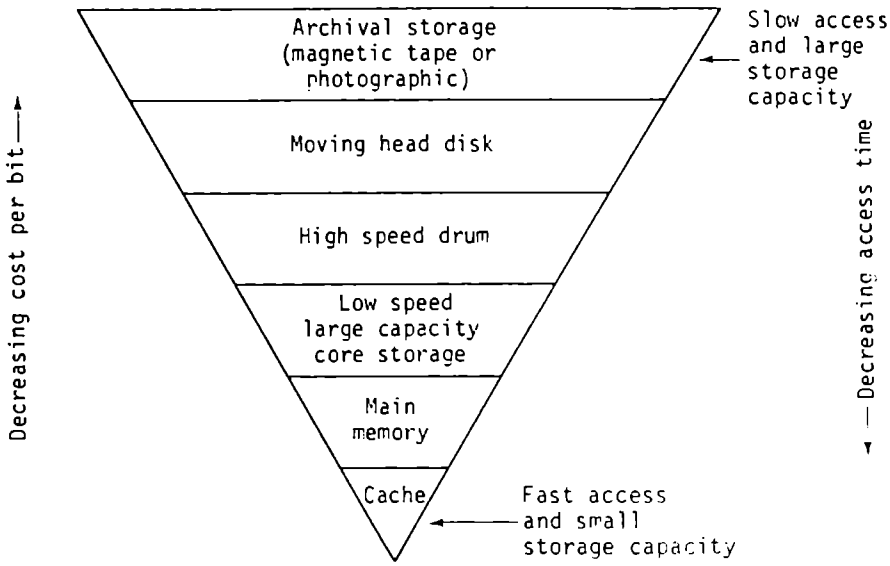


Fig. 5-31. A 6 level memory hierarchy.

to randomly access the cache, 50 sec to randomly access a magnetic tape). Similarly, a large tape library may contain millions or tens of millions of times as many bits as the cache. The cost per bit of storage also varies from level to level, being most expensive for the cache and cheapest for the archival storage.

The advantage of a memory hierarchy is that a huge virtual address space can be provided at an effective access time only slightly worse than that of the fastest level. If the hit ratio (fraction of main memory references that can be satisfied out of the cache) is high, the program will run very fast. In other words, if the cache working set changes slowly, the program will, in effect, be running out of the cache and not out of main memory.

5.5.9. Segmentation

The virtual memory discussed above is one dimensional because the virtual addresses go from zero to some maximum address, one address after another. There are many problems for which having two or more separate virtual address spaces may be much better than having only one. For example, a compiler has many tables that are built up as compilation proceeds, including

1. The source text being saved for the printed listing.
2. The symbol table, containing the names and attributes of declared variables.
3. The numeric constant table, containing all the integer and floating-point constants used.

4. The parse tree, containing the syntactic analysis of the program.
5. The stack used for procedure calls within the compiler.

Each of the first four tables grows continuously as compilation proceeds. The last one grows and shrinks in unpredictable ways during compilation. In a one-dimensional memory, these five tables would have to be allocated contiguous chunks of virtual address space, as in Fig. 5-32.

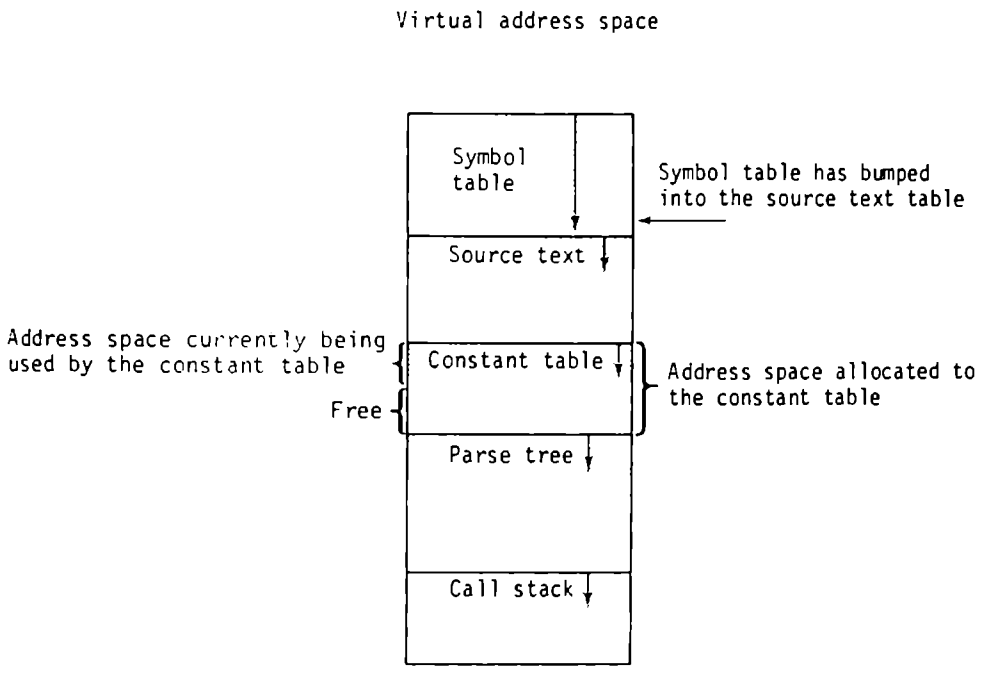


Fig. 5-32. In a 1-dimensional address space with growing tables, one table may bump into another.

Consider what happens if a program has an exceptionally large number of variables. The chunk of address space allocated for the symbol table may fill up, but there may be lots of room in the other tables. The compiler could, of course, simply issue a message saying that the compilation cannot continue because of too many variables, but doing so does not seem very sporting when the other tables still have unused space.

Another possibility is to play Robin Hood, and take space from the tables with a great deal of room and give it to the tables with little room. This shuffling can be done, but it is analogous to managing one's own overlays—a nuisance at best and a great deal of work at worst.

What is really needed is a way of freeing the programmer from having to manage the expanding and contracting tables himself, in the same way that virtual memory freed him of having to worry about organizing his program into overlays.

A straightforward and extremely general solution is to provide the level 3 machine with many, completely independent address spaces, called **segments**. Each segment consists of a linear sequence of addresses, from 0 to some maximum. The length of each segment may be anything from 0 to the maximum allowed. Different segments may, and usually do, have different lengths. Moreover, segment lengths may change during execution. The length of a stack segment may be increased whenever something is pushed onto the stack and decreased whenever something is popped off the stack.

Because each segment constitutes a separate address space, different segments can grow or shrink independently, without affecting each other. If a stack in a certain segment needs more address space to grow, it can have it, because there is nothing else in its address space to bump into. Of course, a segment can fill up, but segments are usually very large and so this occurrence is rare. In order to specify an address in this segmented or two-dimensional memory, the program must supply a two-part address, a segment number, and an address within the segment. Figure 5-33 illustrates a segmented memory being used for the compiler tables discussed earlier.

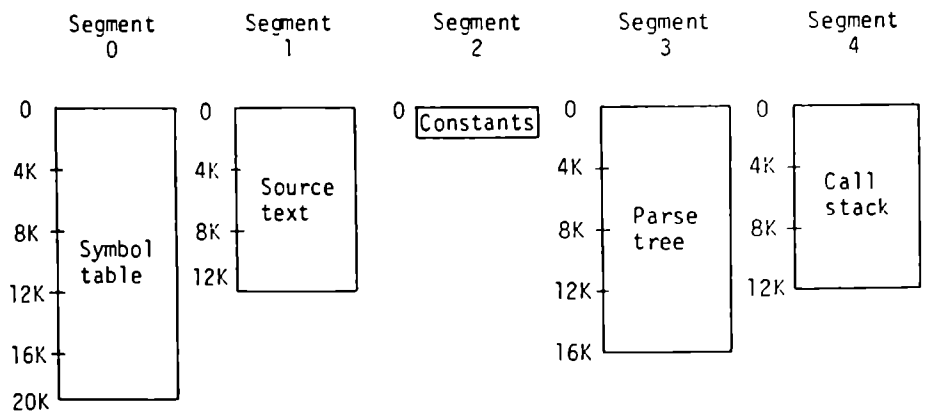


Fig. 5-33. A segmented, or 2-dimensional memory allows each table to grow or shrink independently of the other tables.

We emphasize that a segment is a logical entity, which the programmer is aware of and uses as a single logical entity. A segment might contain a procedure, or an array, or a stack, or a collection of scalar variables, but usually it does not contain a mixture of different types.

A segmented memory has other advantages besides simplifying the handling of data structures that are growing or shrinking. If each procedure occupies a separate segment, with address 0 as its starting address, the linking up of procedures compiled separately is greatly simplified. After all the procedures that constitute a program have been compiled and linked up, a procedure call to the procedure in segment N will use the two-part address (N, 0) to address word 0 (the entry point).

If the procedure in segment N is subsequently modified and recompiled, no other procedures need be changed (because no starting addresses have been modified),

even if the new version is larger than the old one. With a one-dimensional memory, the procedures are packed tightly next to each other, with no address space between them. Consequently, changing one procedure's size can affect the starting address of other, unrelated procedures. This, in turn, requires modifying all procedures that call any of the moved procedures, in order to incorporate their new starting addresses. If a program contains hundreds of procedures, this process can be costly.

Segmentation also facilitates sharing procedures or data between several processes. If a computer has several level 3 machines running in parallel (either true or simulated parallel processing), all of which use certain library procedures, it is wasteful of main memory to provide each one with its own private copy. By making each procedure a separate segment, they can be shared easily, thus eliminating the need for more than one physical copy of any shared procedure to be in main memory.

Since each segment forms a logical entity of which the programmer is aware, such as a procedure, or an array, or a stack, different segments can have different kinds of protection. A procedure segment could be specified as execute only, prohibiting attempts to read from it or store into it. A floating-point array could be specified as read/write but not execute, and attempts to jump to it would be caught. Such protection is very helpful in catching programming errors.

You should try to understand why protection makes sense in a segmented memory but not in a one-dimensional paged memory. In a segmented memory, the user is aware of what is in each segment. Normally a segment would not contain a procedure

Consideration	Paging	Segmentation
Need the programmer be aware that this technique is being used?	No	Yes
How many linear address spaces are there?	1	Many
Can the total address space exceed the size of physical memory?	Yes	Yes
Can procedures and data be distinguished and separately protected?	No	Yes
Can tables whose size fluctuates be accomodated easily?	No	Yes
Is sharing of procedures between users facilitated?	No	Yes
Why was this technique invented?	To get a large linear address space without having to buy more physical memory	To allow programs and data to be broken up into logically independent address spaces and to aid sharing and protection

Fig. 5-34. Comparison of paging and segmentation.

and a stack, for example, but one or the other. Since each segment contains only one type of object, the segment can have the protection appropriate for that particular type.

The contents of a page are, in a sense, accidental. The programmer is unaware of the fact that paging is even occurring. Although putting a few bits in each entry of the page table to specify the access allowed is possible, the programmer, in order to utilize this feature, would have to keep track of where in his address space the page boundaries were, which is precisely the sort of administration that paging was invented to eliminate. Since the user of a segmented memory has the illusion that all segments are in main memory all the time—that is, he can address them as though they were—he can protect each segment separately, without having to be concerned with the administration of overlaying them.

Before going on to the sections on implementation of segmented memories, you should be sure to understand the differences between paging and segmentation. They are summarized in Fig. 5-34.

5.5.10. Virtual Memory on the PDP-11

In this section we will examine the virtual memory on the PDP-11/45, a member of the PDP-11 series. The PDP-11/45 uses a simple form of segmentation. The virtual address space consists of 16 segments, each of which can have a maximum size of 2^{13} bytes. (See Fig. 5-35.) Eight of the segments are for procedures and eight for data, with the "hardware" automatically using procedure segments to fetch instructions and data segments to fetch and store data. In this manner, 2^{17} addresses can be specified in 16 bits.

At first, this fact may appear to violate what we said earlier about only having 2^{16} addresses accessible with 16 bits, but you should note that it is impossible to jump

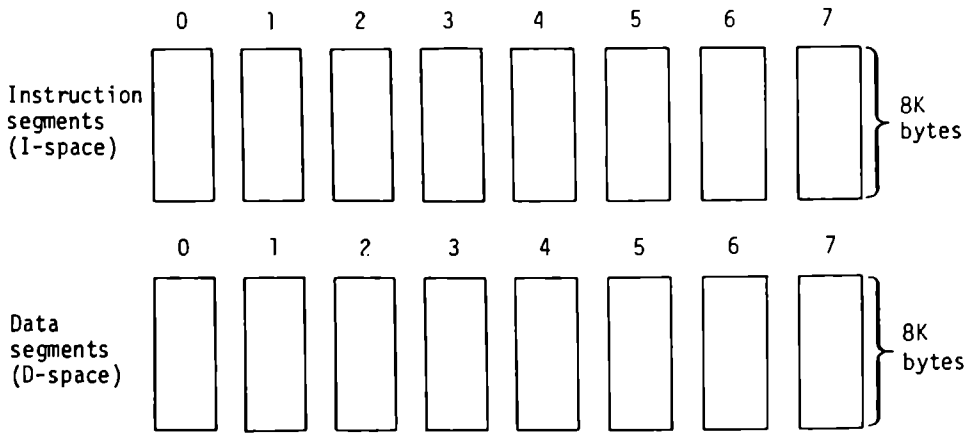


Fig. 5-35. The PDP-11/45 virtual memory has eight instruction segments and eight data segments.

to the data. The instruction

```
JMP 200      (segment 0, address 200)
```

will always jump to the instruction space and not to the data space. Similarly, the instruction

```
CLR 200      (segment 0, address 200)
```

will always clear (set to zero) the word at address 200 of the data space and not the instruction space. There is no way to express the instruction "jump to segment 0 address 200 of the data space," or the instruction "clear the word at address 200 of segment 0 of the instruction space." Half the addressing power has been traded for twice as much addressable memory. In addition to allowing 2^{17} bytes of memory to be addressed in 16 bits, programs are prevented from changing themselves, which reduces errors, aids recursion, and eliminates page transfers by increasing the number of clean pages. Furthermore, programs are prevented from accidently jumping into the data, which helps debugging, because wild jumps into the data are notoriously difficult to track down.

The PDP-11/45 virtual address is broken up into two fields—a three-bit segment field and a 13-bit offset within the segment (see Fig. 5-36). Sixteen segment descriptors are kept in high-speed hardware registers, one describing each segment. When an instruction is being fetched, the three-bit segment field selects one of the eight instruction space descriptors. Similarly, when a data word is being referenced, one of the eight data space descriptors is selected.

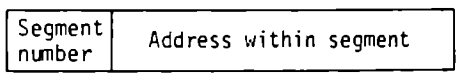


Fig. 5-36. PDP-11/45 virtual addresses are divided into a 3-bit segment number and a 13-bit address within the segment.

A segment descriptor is 32 bits long, as shown in Fig. 5-37. When a particular segment is not currently resident in main memory, it is indicated by having a 0 in the protection field. When a segment is resident, this field specifies which kinds of accesses are allowed (e.g., read only, or read and write). For instruction space segments, execute only is the only possibility. The protection field can be set to trap every access, in order to keep statistics on usage.

Stacks on the PDP-11 grow from high addresses to low addresses. Words pushed onto the stack by consecutive instructions will be moved into consecutively lower-numbered word addresses (e.g., addresses 1006, 1004, 1002, 1000). Consequently, if segment 0 contains a 256-word (512-byte) stack, it will be located in virtual addresses 7680 to 8191 rather than virtual addresses 0 to 511. The stack field of the segment descriptor indicates whether the segment is a stack or not.

The dirty bit is automatically set whenever a segment is modified. The activity bit is automatically set whenever a segment is referenced, whether for reading, or writing, or executing. This bit can be used by the operating system to help determine which page has been least recently used. The segment length field specifies the seg-

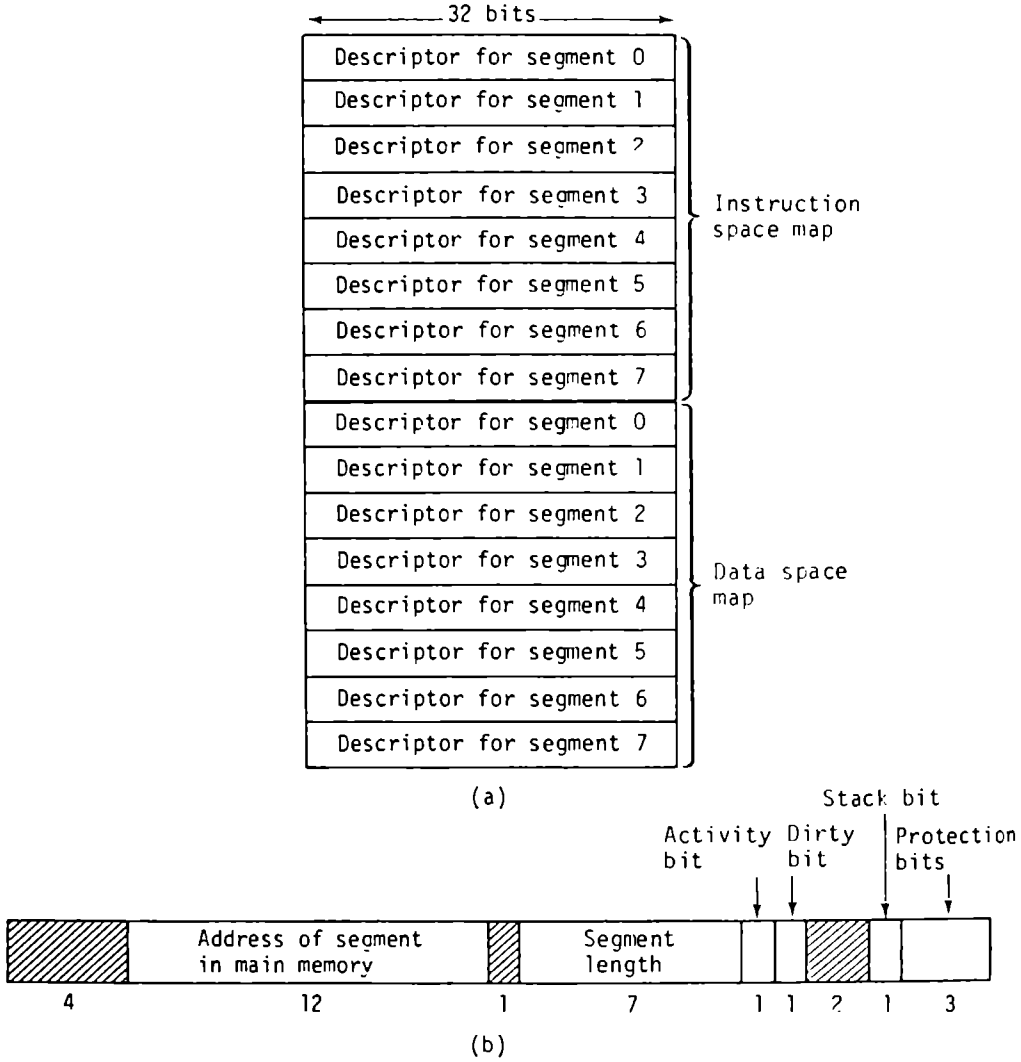


Fig. 5-37. (a) The PDP-11/45 virtual memory map. (b) The fields within a descriptor. The numbers give the field lengths.

ment length. Because a segment can be as long as 2^{13} bytes, 13 bits are needed. To save space in the segment descriptor, the low-order six bits are not stored but are assumed to be 0 instead. Thus segment length is restricted to integral multiples of 64 bytes.

The address field specifies the location of the segment in main memory. To save space in the segment descriptor again, the low-order six bits are not stored here either but are also assumed to be 0. This means that a segment must be positioned at an address that is an integral multiple of 64 bytes. The segment's secondary memory address is kept in a table in main memory.

A PDP-11/45 virtual address is converted to a physical memory address in the following steps:

1. The segment descriptor is fetched, using the three-bit segment number to select a descriptor from the appropriate map.
2. A check is made to see if the segment is in main memory.
3. A check is made to see if there is a protection violation (e.g., an attempt to write into a read only segment).
4. A check is made to see if the offset is within the segment by comparing it to the segment length field.
5. The offset is added to the 18-bit segment origin to give a physical main memory address.

Three different faults can occur during the conversion of a virtual address to a physical address:

1. Segment fault —the segment is not in main memory (or is nonexistent).
2. Length fault —an address beyond the end of the segment has been referenced.
3. Protection fault—the attempted access is not permitted.

If a segment fault occurs, there is a trap to the segment fault handler, whose function is to bring the segment in. The algorithms for deciding which segment to throw out are similar to those used in paged memories, except that each segment has its own length. Consequently, if memory is full of segments smaller than the one referenced, it may be necessary to evict not one but several segments. Furthermore, if a segment grows, it may be necessary to move the other segments in memory around, in order to provide it with enough room.

Checkerboarding

Consider what happens to the memory of Fig. 5-38(a) if segment 1 is evicted and segment 7, which is smaller, is put in its place. We arrive at the memory configuration of Fig. 5-38(b). Between segment 7 and segment 2 is an unused area—that is, a hole. Then segment 4 is replaced by segment 5, as in Fig. 5-38(c), and segment 3 is replaced by segment 6, as in Fig. 5-38(d). After the system has been running for a while, memory will be divided up into a number of chunks, some containing segments and some containing holes. This phenomenon is called **checkerboarding**, and, as a result, memory is wasted in the holes.

Consider what would happen if the program referenced segment 3 at the time memory was checkerboarded, as in Fig. 5-38(d). The total space in the holes is 10K, more than enough for segment 3; but because the space is distributed in small, useless pieces, segment 3 cannot simply be loaded. Instead another segment must be removed first.

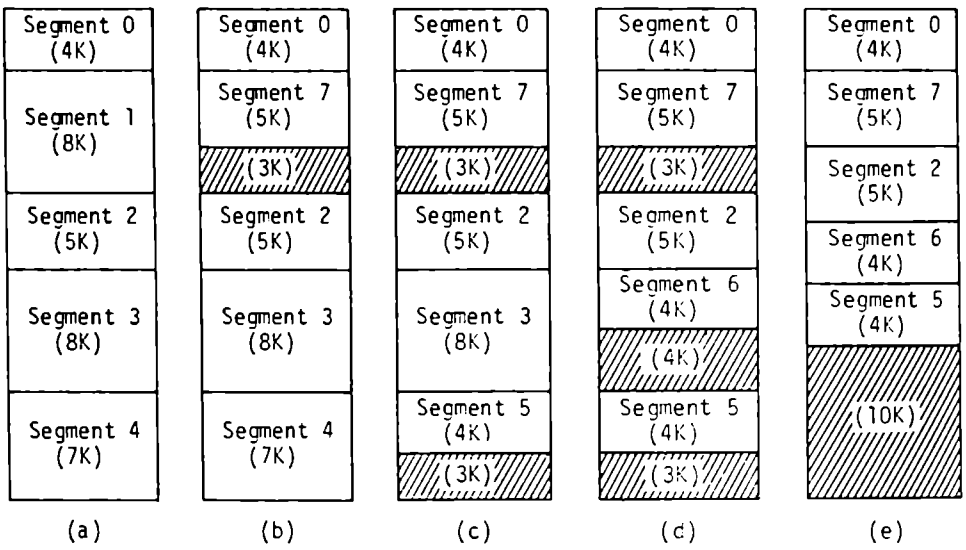


Fig. 5-38. The development of a checkerboarded memory. (e) Removal of the checkerboarding in (d) by compacting.

One way to avoid checkerboarding is as follows: every time a hole appears, move the segments following the hole closer to memory location 0, thereby eliminating that hole but leaving a big hole at the end. Alternately, one could wait until the checkerboarding became quite serious (e.g., more than a certain percentage of the total memory wasted in holes) before performing the compaction. Figure 5-38(e) shows how the memory of Fig. 5-38(d) would look after compaction. The intention of compacting memory is to collect all the small useless holes into one big hole, into which one or more segments can be put. Compacting has the obvious drawback that some time is wasted doing the compacting. It is usually too time-consuming to compact after every hole is created.

If the time required for compacting memory is unacceptably large, an algorithm is needed to determine which hole to use for a particular segment. Hole management requires maintaining a list of the addresses and sizes of all holes. One popular algorithm, called **best fit**, chooses the smallest hole into which the needed segment will fit. The idea is to match holes and segments so as to avoid breaking off a piece of a big hole, which may be needed later for a big segment.

Another popular algorithm, called **first fit**, scans the hole list and chooses the first hole big enough for the segment to fit into. Doing so obviously takes less time than checking the entire list to find the best fit. Surprisingly, first fit is also a better algorithm in terms of overall performance than best fit, since the latter tends to generate a great many very small, totally useless holes (Knuth, 1974).

Both first fit and best fit have a tendency to decrease hole size. Whenever a segment is placed in a hole bigger than itself, which happens almost every time a segment is read in (exact fits are rare), the hole is divided into two parts. One part is occupied

by the segment, and the other part is the new hole. The new hole is always smaller than the old hole. Unless there is a competing process recreating big holes out of small ones, both first fit and best fit will eventually fill memory with small useless holes.

One such competing process is the following one. Whenever a segment is removed from memory and one or both of its nearest neighbors are holes, rather than segments, the adjacent holes can be coalesced into one big hole. If segment 5 were removed from Fig. 5-38(d), the two surrounding holes and the 4K used by the segment would be merged into a single 11K hole.

With segments as small as the PDP-11's, the full power of segmentation is not really used. In fact, one could legitimately regard the PDP-11/45 as having two paged address spaces (I-space and D-space) with a page size of 8K bytes, rather than a segmented address space. The fact that the units of address space are variable sized and can be individually protected makes them more like segments than pages, however.

If you are confused about this point, you may take comfort in the fact that you are not alone. Prior to 1973, DEC publications described the PDP-11/45's virtual memory in terms of segments; and starting in 1973 it was described in terms of pages, although the pre-1973 hardware was identical to the post-1973 hardware. The following sections describe segmentation on larger computers, where the distinction between paging and segmentation is much clearer.

5.5.11 The MULTICS Virtual Memory

The ideal virtual memory consists of a very large number of very large segments. The MULTICS operating system for the Honeywell 6180 provides each program with a virtual memory of up to 2^{18} segments (more than 250,000), each of which can be up to 65,536 words (36 bits) long. To implement this virtual memory, the MULTICS designers choose to treat each segment as a virtual memory and to page it, combining the advantages of paging (uniform page size and not having to keep the whole segment in memory if only part of it is being used) with the advantages of segmentation (ease of programming, modularity, protection, and sharing).

Each MULTICS process has a segment table, with one descriptor per segment. Since there are potentially more than a quarter of a million entries in the table, the segment table is itself a segment and is paged. A segment descriptor contains an indication of whether the segment is in main memory or not. If any part of the segment is in memory, then the segment is considered to be in memory and its page table will be in memory. If the segment is in memory, then its descriptor contains a pointer to its page table [see Fig. 5-39(a)]. Because physical memory addresses are 24 bits, the low-order six bits of the address are assumed to be 0. The descriptor also contains the segment size, the protection bits, and a few other items. Figure 5-39(b) illustrates the MULTICS segment descriptor. The address of the segment in secondary memory is not in the segment descriptor but in another table used by the segment fault handler.

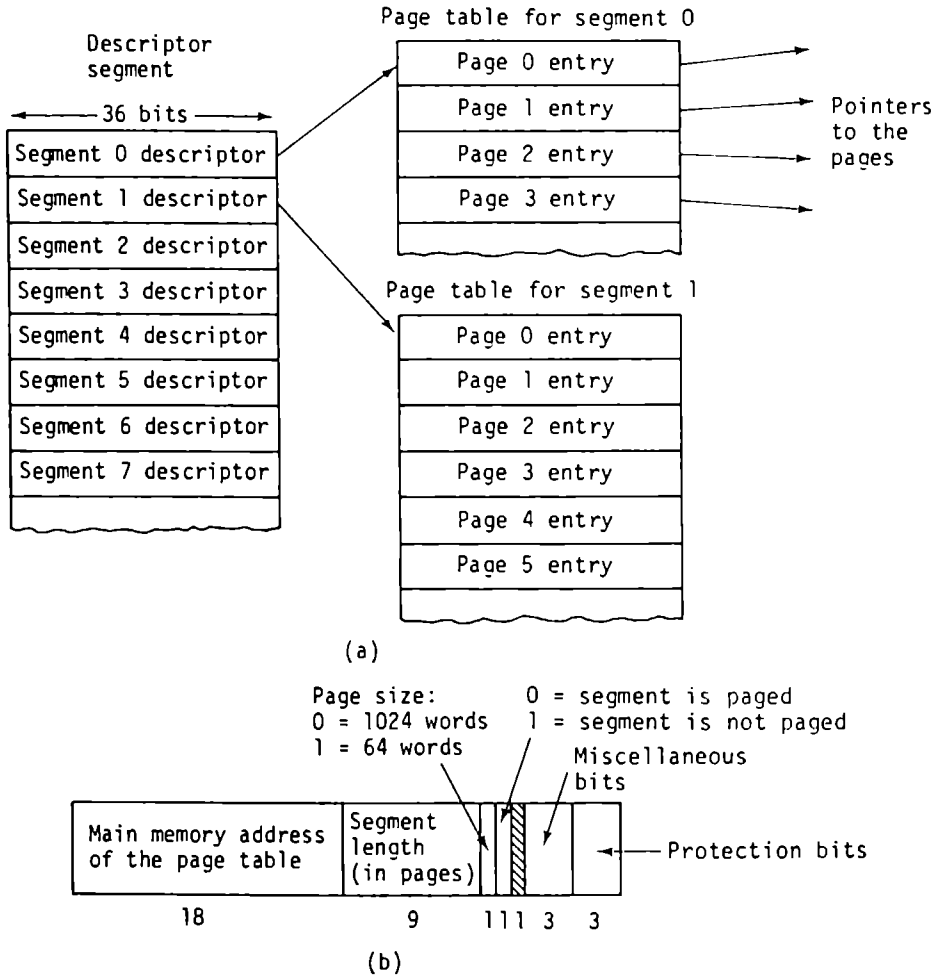


Fig. 5-39. The MULTICS virtual memory. (a) The descriptor segment provides pointers to the page tables, which in turn point to the pages. (b) A MULTICS segment descriptor. The numbers are the field lengths.

Each segment is an ordinary virtual address space and is paged in the same way as the nonsegmented paged memory described in Section 5.5.2. The normal page size is 1024 words (although a few small segments used by MULTICS itself are not paged or are paged in units of 64 words).

An address in MULTICS consists of two parts: the segment and the address within the segment. The address within the segment is further divided into a page number and a word within the page, as shown in Fig. 5-40. When a memory reference occurs, the following algorithm is carried out.

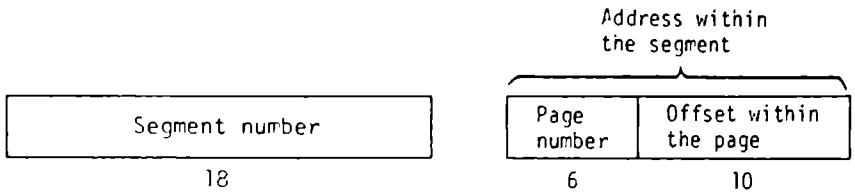


Fig. 5-40. A MULTICS virtual address consists of an 18-bit segment number and a 16-bit address within the specified segment. The segment address is further dividend into a page and an offset within the page.

ALGORITHM

1. The segment number is used to find the segment descriptor.
2. A check is made to see if the segment's page table is in memory. If it is not, a segment fault occurs. If there is a protection violation, a fault (trap) occurs. If the page table is in memory and the access allowed, the page table is located.
3. The page table entry for the requested virtual page is examined. If the page is not in memory, a page fault occurs. If it is in memory, the main memory address of the start of the page is extracted from the page table entry.
4. The offset is added to the page origin in order to give the main memory address where the word is located.
5. The read or store finally takes place.

This process is illustrated schematically in Fig. 5-41. For simplicity, the fact that the descriptor segment is itself paged has been omitted. What really happens is that a register, called the descriptor base register, is used to locate the descriptor segment's page table, which, in turn, points to the pages of the descriptor segment. Once the descriptor for the needed segment has been found, the addressing proceeds as in Fig. 5-41.

As you have no doubt guessed by now, if the preceding algorithm were actually carried out by the operating system on every instruction, programs would not run very fast. In reality, there is a 16-word high-speed associative memory in the MULTICS hardware, as shown in Fig. 5-42. When an address is presented to the computer, the addressing hardware first checks to see if the virtual page is in the associative memory. If so, it gets the page frame number directly from the associative memory and forms the actual address of the referenced word without having to look in the descriptor segment or page table.

The 16 most recently referenced pages are kept in the associative memory. Programs whose working set is smaller than the size of the associative memory will come to equilibrium with the addresses of the entire working set in the associative memory and therefore will run efficiently. If the page is not in the associative memory, the descriptor and page tables are actually referenced to find the page frame address, and the associative memory is updated to include this page, the least recently used

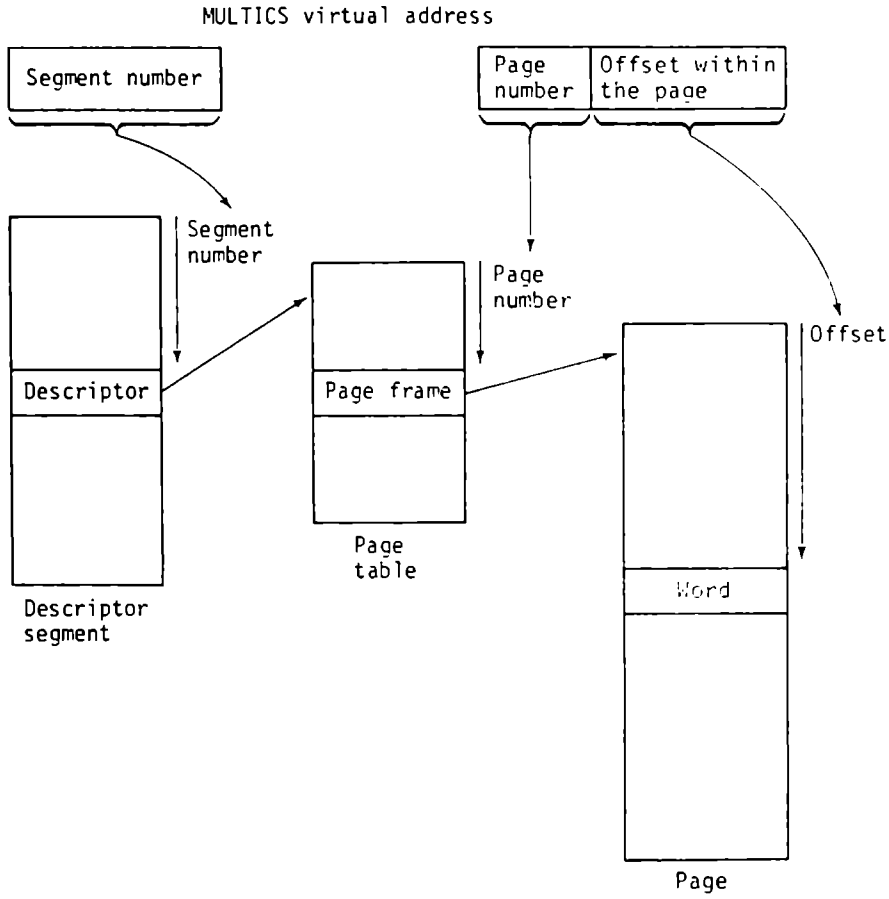


Fig. 5-41. Conversion of a two-part virtual address into a main memory address in MULTICS.

Comparison field					Is this entry used?
Segment number	Virtual page	Page frame	Protection	Age	
4	1	7	Read/write	13	1
6	0	2	Read only	10	1
12	3	1	Read/write	2	1
					0
2	1	0	Execute only	7	1
2	2	12	Execute only	9	1

Fig. 5-42. A simplified version of the MULTICS associative memory. The existence of two page sizes makes the actual associative memory more complicated.

page being thrown out. The age field keeps track of which entry has been least recently used. The reason that an associative memory is used is that the segment and page number of all the entries must be compared simultaneously, for speed.

5.5.12. Virtual Memory on the IBM 370

Like MULTICS, the 370 has a sophisticated virtual memory with paged segments. IBM calls virtual memory **dynamic address translation**. The 370's total virtual address is smaller then that of MULTICS (2^{24} 8-bit bytes versus 2^{34} 36-bit words). Due to the comparatively small address space, the division of the 24 bits into segment number and address within the segment is much more critical than in MULTICS. A decision to have 2^{18} segments like MULTICS would have meant a maximum segment size of 64 bytes, hardly a realistic possibility.

In order to provide additional flexibility within the limitations of a 24-bit virtual address, the designers of the 370 provided two different possibilities:

16 segments of 2^{20} bytes (262,144 words)

256 segments of 2^{16} bytes (16,384 words)

The MULTICS goal of making each procedure and each large array a separate segment will be difficult to realize on the 370 if there are more than 16 procedure and data segments, some of which exceed 16K words.

All segments on the 370 are paged. Two page sizes are provided: 2K bytes and 4K bytes. The 370 uses a descriptor segment and page tables, similar to those used in MULTICS. Figure 5-43 shows the formats of the segment descriptors and page table entries. The segment length is in units of $\frac{1}{16}$ of the maximum number of pages, using zero-origin counting. The page table address is supplied with three low-order zeros, to give a 24-bit main memory address. Similarly, the page addresses are only the high-order bits of a 24-bit address.

The addressing mechanism on the 370 is the same one used by MULTICS, including the use of a small, high-speed associative memory. The size of the associative memory is different for models of the 370 series, being larger in the larger machines. Associated with each page are bits to record whether or not the page has been used or modified. These bits, along with the protection bits, are stored not in the page or segment tables but in special seven-bit registers, called storage keys.

Segmentation on the 370 differs from that of MULTICS in one important way. If a 370 base register contains the highest address of a segment and 1 is added to it, it will then point to address 0 of the following segment. Because each segment is a logical entity, unrelated to the segments with segment numbers above and below its own, the property of incrementing a register and having it suddenly point to a new and unrelated segment is highly undesirable. On MULTICS, no operation on the page and offset part of the virtual address can change the segment part; that is, there is no "carry" from it into the segment part. The PDP-11 suffers from the same unfortunate "carry" problem as the 370.

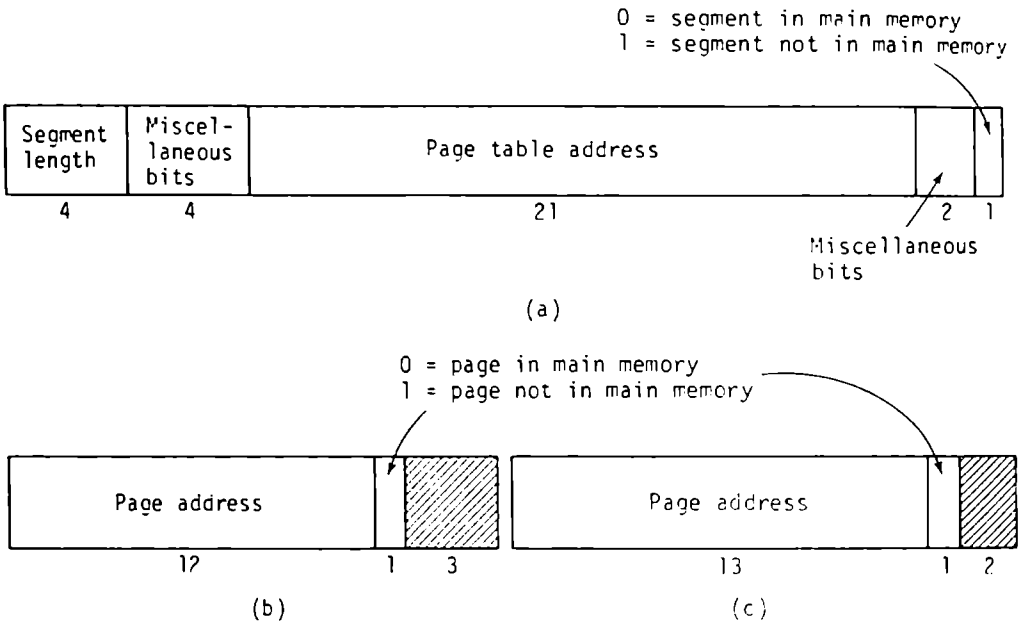


Fig. 5-43. (a) A 370 segment descriptor. (b) A page table entry for a segment with 4K byte pages. (c) A page table entry for a segment with 2K byte pages.

5.5.13. Segmented Virtual Memory and File I/O

The availability of a very large segmented address space has important implications for file i/o. By making each file into a segment, a program can directly reference any part of any file without the need for special virtual i/o instructions. For example, a program could directly add an integer in a file to a register, using the ordinary ADD instruction, without the necessity of first "reading in" the integer. If the logical record referenced were not in core, a page fault would occur, and the page fault handler would bring it in.

In a sense, virtual i/o instructions on files and virtual memory are competing mechanisms for transferring information from secondary memory to main memory. Virtual memory is by far the more convenient of the two from the programmer's point of view, because it does not require him to issue any special instructions to access his data, or even be aware of whether the data is in main memory or not. Furthermore, it is wasteful to have two distinct mechanisms in the operating system, both of which perform the same function (transferring information from secondary to main memory).

The MULTICS system carries the correspondence between files and segments to its logical extreme: no distinction is made between files and segments. When a process is started up, all the files it initially needs (both procedures and data) are given segment numbers in order to make them directly addressable without the need for any i/o

instructions. If it is discovered later that other files from one of the user's directories (or someone else's directories, if permitted) are needed, they will automatically be added to the virtual address space by simply indirectly addressing their symbolic names. This technique is called dynamic linking and is described in Section 6.5.4. Because all the parts of all the files that a program needs can be accessed directly simply by using the proper virtual address, a typical MULTICS program never does any explicit i/o, even if it is processing large amounts of data.

Although files and segments are not identical on the 370, it is possible to make a file addressable as a segment by using the Virtual Storage Access Method (VSAM). If accessing a number of large files by mapping them onto segments is desired, the total amount of virtual address space available becomes important. The total virtual address space on the 370 is approximately enough to access the data contained on half of one magnetic tape. In contrast, the MULTICS virtual address space is sufficient to access all the data on over 1000 full magnetic tapes.

5.6. JOB CONTROL LANGUAGES

Besides implementing a wide variety of virtual instructions for the level 3 programmer to use, nearly all operating systems provide an additional method for users to take advantage of the facilities provided by the operating system. This method takes the form of a **job control language**, or JCL. The control cards with an asterisk in column 1 of Fig. 1-3 form a very simple job control language. The commands available in most job control languages are things like

1. compile a program using a certain compiler.
2. link up all the procedures used in a program.
3. execute a linked-up program.
4. define a file.
5. enter a file in a directory.
6. remove a file from a directory.
7. request a magnetic tape to be mounted.
8. dump memory in octal or hexadecimal.
9. make a copy of a file.
10. save the state of a process for continuation later (checkpoint).

The job control languages provided by some operating systems are extremely complex and allow sophisticated users to do sophisticated things. Entire books have been written trying to explain the IBM 370 JCL (Cadow, 1970, for example). Most job control languages resemble assembly langages, with a symbolic opcode followed by a few operands on each line. An example of a simple job control language program

for the Cyber 70 is as follows:

EXAMPLE, CM120000, T300,NT1.	job card
ACCOUNT(28474)	accounting
ATTACH(PROGRAM, ID = AST)	get access to permanent file
PLI(I = PROGRAM)	compile program
RFL(60000)	reduce memory to 60000 words
ATTACH(DATA,ID = AST)	get access to permanent file
REQUEST(RESULTS,*PF)	request new file on permanent file disk
REQUEST(TAPE1,NT,VSN = S4004B)	request tape to be mounted
LGO(DATA,TAPE1,RESULTS)	execute program
CATALOG(RESULTS,ID = AST)	enter RESULTS in permanent file directory
REWIND(RESULTS)	rewind file
COPY(RESULTS,PUNCH)	punch results on cards as well
UNLOAD(TAPE1)	rewind and unload the tape

Strangely, manufacturers who provide a dozen or more programming languages for the user to choose from often provide only one job control language. Worse yet, this arbitrary decision to force all programmers to use the same job control language is totally unnecessary. A much better solution is for the operating system to implement level 3 instructions for all the functions now provided by the job control cards. A user-supplied interpreter running on the level 3 machine could then read job control language statements, examine them, and execute the proper level 3 instruction or instructions to carry them out. This process would make the job control language simply another level in the hierarchy.

The advantage of this method is that since the job control language interpreter runs on the level 3 machine, and is not buried deep within the operating system (which runs on the level 2 machine), a computer center could provide a whole library of different interpreters. Simple job control languages could be available for beginners, and more sophisticated job control languages could be available for people with complicated jobs. Furthermore, since any programmer may write his own level 3 programs, anyone who needed a special job control language for a certain application could simply write his own interpreter for it.

In contrast, the job control language interpreters for the 370, Cyber 70, and PDP-11 are all part of their respective operating systems and thus cannot be modified or replaced by the users. The important issue is not which level the interpreter runs on but whether or not that level is also available to the users so that they can provide their own interpreters if desired.

PROBLEMS

1. Explain the following in your own words:
   a. file
   b. indexed sequential file
   c. circular buffer
   d. race condition
   e. page fault
   f. segment fault
   g. semaphore
   h. deadlock
2. Why does an operating system only interpret some of the level 3 instructions, whereas a microprogram interprets all the level 2 instructions?
3. Why do many file systems require that a file be explicitly opened with an OPEN instruction before being used?
4. A sequential file and a random access file are both allocated as a series of consecutive sectors. What is the difference
   a. to the level 3 program?
   b. to the level 2 program (the operating system)?
5. Compare the bit map with the hole list method for keeping track of free disk space for a disk with 200 cylinders, each containing 20 tracks of 16 sectors. Does the number of bits required for either method depend on the number of free sectors on the disk? How many holes would it take before the hole list was bigger than the bit table? Assume that the allocation unit is the sector. The word length is 32 bits, and a hole can be represented in one word.
6. On a certain computer, a level 3 program can create as many files as it needs, and all files may grow dynamically during execution, without giving the operating system any advance information about their ultimate size. Do you think files are stored in consecutive sectors or not? Explain.
7. Make a list of all the virtual i/o instructions provided by your operating system.
8. Consider the following method by which an operating system might implement semaphore instructions. Whenever the CPU was about to UP or DOWN a semaphore (an integer variable in memory), it first sets the CPU priority bits or the mask bits in such a way that no i/o device (including the clock) could interrupt it. Then it fetches the semaphore, tests it, branches accordingly, and, finally, resets the priority or mask bits to be interruptable again. Does this method prevent race conditions if
   a. there is a single CPU that simulates parallel processing by switching from one process to another every 100 msec?
   b. there are two CPUs that share a common memory in which the semaphore is located?
9. Draw a diagram showing which of the processes P1, P2, and P3 are running and which are blocked from $t = 0$ to $t = 1000$. All three processes perform UP and DOWN instructions on the same semaphore. When two processes are potential candidates for

being restarted, P1 gets preference over P2 and P3, and P2 gets preference over P3. At $t = 0$, all three are running and the semaphore is 1.

$t = 100$ P1 does a DOWN
$t = 200$ P1 does a DOWN
$t = 300$ P2 does an UP
$t = 400$ P3 does a DOWN
$t = 500$ P1 does a DOWN
$t = 600$ P2 does an UP
$t = 700$ P2 does a DOWN
$t = 800$ P1 does an UP
$t = 900$ P1 does an UP

10. The Nevercrash Operating System Company has been receiving complaints from some of its customers about its latest operating system, which includes semaphore operations. The irate customers feel that it is immoral for a process to block (they call it "sleeping on the job"). Since it is company policy to give the customers what they want, it has been proposed to add a third semaphore operation PEEK, in addition to UP and DOWN. PEEK simply examines the value of a semaphore, without changing it or blocking the process. In this way, programs that feel it is immoral to block can check on the value of a semaphore to see if it is safe to execute a DOWN instruction. Will this idea work if only two processes are using the semaphore? If there are three or more? Explain.

11. In an airline reservation system, it is necessary to ensure that while one process is busy using a file, no other process can also use it. Otherwise two different processes, working for two different ticket agents, might each inadvertently sell the last seat on some flight. Suppose that the level 3 machine being used has semaphore instructions. Devise a method for guaranteeing that once a process has begun using a file, no other process will try to use it (assuming that no process cheats and just uses the file without regard to whether it is supposed to or not).

12. Consider two processes that each create very large scratch files on a shared minidisk. If no process needs more disk space than is available, is that sufficient to guarantee that no deadlock can occur?

13. Three methods for handling deadlocks are as follows:

a. Require all processes to request all resources before beginning execution.
b. When process P needs some resource currently possessed by process Q, simply take it away from Q and give it to P.
c. Don't take any precautions, but when a deadlock occurs, just have the operator rerun all the jobs.

For each method, describe one disadvantage or one situation where it does not work.

14. Two processes are competing for a tape drive and a plotter. Each process will execute 1 million instructions before it has finished. The state of the computation can be described by a graph in which the x axis is the number of instructions P1 has executed and the y axis is the number of instructions P2 has executed. For example, the point (500, 1 000) means that P1 has already executed 500 instructions and P2 has already executed 1000 instructions. The system has only one CPU. A horizontal line indicates that P1 is running, and a vertical line indicates that P2 is running. Diagonal lines are

not allowed, because both processes cannot be making progress simultaneously on a system with only one CPU. A successful run will have a connected series of line segments beginning at (0, 0) and ending at (1 000 000, 1 000 000). As an example, one possible run is given by connecting the points (0, 0), (0, 1 000), (2 000, 1 000), (2 000, 800 000), (500 000, 800 000), (500 000, 1 000 000), (1 000 000, 1 000 000).

P1 needs the tape drive between instructions 300000 and 700000 and the plotter between 500000 and 800000. P2 needs the tape drive between 400000 and 600000 and the plotter between 300000 and 500000. Once a process no longer needs a device, it immediately returns it. Draw a diagram representing this information. Find the regions that are impossible—that is, that the computation state can never enter (e.g., both processes in possession of the plotter). A region exists that can be entered but from which there is no exit. Show this region on your diagram.

15. Show the values of IN and OUT for a circular buffer of 65 words after each of the following operations:
    a. 22 words are put in
    b. 9 words are removed
    c. 40 words are put in
    d. 17 words are removed
    e. 12 words are put in
    f. 45 words are removed
    g. 8 words are put in
    h. 11 words are removed

16. For each of the following situations, tell whether an ordinary unpaged unsegmented memory, an unsegmented paged memory, or a paged and segmented memory is most appropriate:
    a. a program with four highly dynamic stacks
    b. a program with one huge linear data base
    c. a compiler with five growing tables
    d. a program that inverts a small matrix in place

17. Devise an algorithm that chooses segments to remove for an unpaged segmented memory, such as the PDP-11/45's. Consider the case where several segments must be removed to make room for a big segment.

18. Compare fragmentation with checkerboarding. What can be done to alleviate each?

19. A virtual memory has a page size of 1024 words. There are eight virtual pages and 4096 words of physical memory. The page table is

Virtual Page	Page Frame
0	3
1	1
2	not in main memory
3	not in main memory
4	2
5	not in main memory
6	0
7	not in main memory

a. Make a list of all virtual addresses that will cause page faults if used.
b. Compute the main memory address, if any, corresponding to each of the following virtual addresses: 0, 3728, 1023, 1024, 1025, 7800, 4096.

20. A segmented memory has paged segments. Each virtual address has a 2-bit segment number, a 2-bit page number, and an 11-bit word within the page. The main memory contains 32,768 words. Each segment is either read only, read/execute, read/write, or read/write/execute. The page tables and protection are as follows.

Segment 0		Segment 1		Segment 2	Segment 3	
Read Only		Read/Execute		Read/Write/Execute	Read/Write	
Virtual Page	Page Frame	Virtual Page	Page Frame		Virtual Page	Page Frame
0	9	0	On drum	Page table	0	14
1	3	1	0	not in	1	1
2	On drum	2	15	main	2	6
3	12	3	8	memory	3	On drum

a. How many virtual pages are there in the address space?
b. For each of the following, tell what physical address is computed from the virtual address, if any. If a segment, page, or protection fault occurs, specify which one.

Access	Segment	Page	Offset within Page
1. fetch data	0	1	1
2. fetch data	1	1	10
3. fetch data	3	3	2047
4. store data	0	1	4
5. store data	2	1	2
6. store data	1	0	14
7. jump to it	1	3	100
8. fetch data	0	2	50
9. fetch data	2	0	5
10. jump to it	3	0	60

21. In the paged systems discussed in the text, the page fault handler was part of the level 2 machine and thus was not present in any level 3 program's address space. It was tacitly assumed that each level 3 machine had a fixed chunk of main memory associated with it, and that when a page had to be removed, it was removed from the part of the main memory belonging to the program causing the page fault. Another strategy is to allow the page fault handler to remove any page, even a page associated with a different

level 3 machine. If the operating system is free to remove any page, it might conceivably choose the page containing the page fault handler itself.

a. What would happen if a page fault occurred when the page fault handler was not in main memory?

b. Devise a simple method for preventing this situation.

22. Give an advantage and a disadvantage of

a. demand paging.

b. preloading of the working set.

23. When a computer with only one CPU is supporting more than one process, it is necessary to have some algorithm for deciding which process gets the CPU—that is, who runs. Three processes A, B, and C each need 1 sec of computing time. Consider the following scheduling algorithms.

1. Round Robin. First run A for 100 msec, then B for 100 msec, then C for 100 msec, then A again for 100 msec, then B again for 100 msec, etc. until all three processes are finished.

2. Run to Completion. First, let A run until it finishes, then run B until it finishes, and then run C until it finishes.

a. Assume that the first process begins at $t = 0$ and that scheduling and switching take no time. For each scheduling algorithm, at what time do A, B, and C each finish?

b. If all three processes begin waiting for service at $t = 0$, what is the average turnaround time for each algorithm?

c. In a time-sharing system, users expect quick responses to short requests for service, such as editing a program. Do either of these algorithms present problems in this respect?

24. You are trying to decide which drum to buy for use as a secondary memory on a computer with a paged virtual memory. The Big Eight Drum Company has a drum with 64 tracks of 8K words around the circumference. The Kilodrum Company sells a drum with 512 tracks of 1K. Both drums can switch from track to track instantaneously. The Big Eight drum has a 32-msec rotation time, and the Kilodrum has an 8-msec rotation time. Plot the average time to read a block of N words as a function of N for each drum, including both access and read time. Under what conditions would the Big Eight drum be an advantage?

25. The method of making a file accessible as a segment eliminates the need for explicit i/o instructions. Is this more like a random access file or a sequential file? List some advantages and disadvantages of accessing files via the virtual memory compared to explicit file i/o instructions.

26. Here we will consider the problem of sharing procedures between two processes. As an example, consider a square root procedure that is to be simultaneously present in the address space of two processes. On the machine in question, a paged, but not segmented, virtual memory is used. Process A has a main program of 10K words, so the square root procedure is located at virtual address 10K. Process B has a main program of 20K words, so the square root procedure is located at virtual address 20K. The first instruction in the square root procedure tests the actual parameter and jumps to word 100

of the procedure if it is negative, in order to print an error message. The computer has a JUMP NEGATIVE instruction, in which the address to jump to can be specified in various ways. For each of the following addressing methods, specify whether or not it is possible to share the square root procedure. If so, how? If not, why not? The actual parameter is in a register, and the JUMP NEGATIVE instruction tests that register.

a. base register addressing
b. self-relative addressing
c. direct addressing

27. Now consider how the square root procedure of Problem 26 could be shared on a machine with a segmented virtual memory. The square root procedure occupies an entire segment. Process A addresses this segment as segment 3, and process B addresses this segment as segment 7. This machine has two kinds of instructions that reference memory:

a. Instructions that contain a segment number and an address within the instruction itself.
b. Instructions that only contain an address, with the segment number being understood as the segment in which the instruction itself is located.

Which kind of instruction should the JUMP NEGATIVE be? Explain your answer.

28. In a three-level memory hierarchy consisting of a cache, main memory, and a drum, the intention is that nearly all the instructions will be executed out of the cache. Assume that the cache has a cycle time of A nsec and that, if a reference to memory is not in the cache, then B nsec are needed to load the cache with the appropriate block. Furthermore, if the word referenced is not in main memory either, then a drum transfer must be scheduled, costing C nsec of CPU time for the administration, including loading the cache (the drum latency is not included because the CPU can do something else while waiting). If one out of every M instructions requires loading the cache from main memory, and one out of every N instructions requires a drum reference, give a formula expressing the effective instruction time of the computer. Assume that B 100A and C = 100,000A. Plot the effective instruction time as a function of M from M = 10 to M = 1000. Make one curve for N = 10^3 and N = 10^4 and N = 10^5 in units of A. Interpret the curves in words.

29. On a computer with segmentation, each CPU has an associative memory in which to look up heavily used pages. There is a level 2 instruction to erase the associative memory. Whenever the CPU loads a page from the drum, it must alter the page table of the process whose page it is, in order to indicate that the page is now in main memory. To ensure that the associative memory accurately reflects the contents of the page table, the operating system clears the associative memory before restarting the process, so that it will dynamically fill up again with values taken from the updated page table. What problems can arise if the computer has several CPUs and a page may belong to several processes that are running simultaneously on different CPUs?

30. Consider the following LRU algorithm for a cache with N blocks. The cache hardware (or the microprogram) maintains an N × N matrix, with the rows and columns numbered 1 through N. Whenever block N is referenced, row N is set to all 1's. After that step, column N is set to all 0's. At any point in time, the row with the lowest value (read as a binary number) is the least recently used. Show that this algorithm works for a four-block cache and the following sequence of block references: 1234423134344.

31. Write an assembly language procedure to implement random access files with fixed-length logical records accessed by record number. The procedure should have four entry points, with the following parameters:

    OPEN (FILENAME,MAXRECS,RECSIZE)
    READREC (FILENAME,RECNO,BUFFER)
    WRITEREC (FILENAME,RECNO,BUFFER)
    CLOSE (FILENAME)

    where

    FILENAME is the name of the file.
    MAXRECS is the maximum number of records the file may have.
    RECSIZE is the record size.
    RECNO is the record to be read or written (starting at 0).
    BUFFER is the buffer where the record comes from or goes to.

    You may use any features of your operating system to help you. A call to OPEN must precede any reading or writing on a file, but reads and writes may be mixed. Print an error message if an attempt is made to read a record that has not yet been written.

32. Write a function to convert segmented virtual addresses to main memory addresses. The function has six parameters:
    a. the descriptor segment (an array)
    b. the page tables (an array)
    c. the access type (0 = read, 1 = write, 2 = execute)
    d. the segment number
    e. the page number
    f. the word within the page
    Choose segment and page sizes appropriate to your computer, even if it does not have virtual memory. If the virtual address can be converted to a main memory address, return that address; otherwise return an error code as follows.

    −1 = segment does not exist (or is not attached)

    −2 = segment fault (page table not in main memory)

    −3 = segment length fault (page is outside segment)

    −4 = page fault (page is not in main memory)

    −5 = protection fault

33. Make a list of 10 control cards used on your computer. What does each one do?

34. Write an assembly language program that compares two files and prints out whether they are identical or not.

35. Write an assembly language function that returns as its value the Nth word of a previously opened random access file, where N is a parameter.

36. List as many ways as you can to make your operating system crash.

6 THE ASSEMBLY LANGUAGE LEVEL

In Chapters 3, 4, and 5 three different levels were discussed, all of which are present on many contemporary computers. This chapter is primarily concerned with a fourth level that is also present on almost all modern computers: the assembly language level. The assembly language level differs in a significant respect from the microprogramming, conventional machine, and operating system machine levels—it is implemented by translation rather than by interpretation.

Programs that convert a user's program written in some language to another language are called translators. The language in which the original program is written is called the **source language,** and the language to which it is converted is called the **target language.** Both the source language and the target language define levels. If a processor that can directly execute programs written in the source language is available, there is no need to translate the source program into the target language.

Translation is used when a processor (either hardware or an interpreter) is available for the target language but not for the source language. If the translation has been performed correctly, the execution of the translated program on its processor will give precisely the same results as the execution of the source program would have given had a processor for it been available. Consequently, it is possible to implement a new level for which there is no processor by first translating programs written for that level to a target level and then executing the resulting target level programs.

It is important to note the difference between translation, on the one hand, and interpretation on the other. In translation, the original program in the source language is not directly executed. Instead it is converted to an equivalent program called an **object program** or **object module,** whose execution is carried out only after the transla-

tion has been completed. In translation, there are two distinct steps:

1. Generation of an equivalent program in the target language.
2. Execution of the newly generated program.

These two steps do not occur simultaneously. The second step does not begin until the first has been completed. In interpretation, there is only one step: executing the original source program. No equivalent program need be generated first.

While the object program is being executed, only three levels are in evidence: the microprogramming level, the conventional machine level, and the operating system machine level. Consequently, three programs—the user's object program, the operating system, and the microprogram—can be found in the computer's memory at run time. All traces of the original source program have vanished. Thus the number of levels present at execution time may differ from the number of levels present before translation. It should be noted, however, that although we define a level by the instructions and linguistic constructs available to its programmers (and not by the implementation method), other authors may make a greater distinction between levels implemented by execution-time interpreters and levels implemented by translation. In other words, a viewpoint in which only the run-time levels are counted as "levels" is also possible.

6.1. INTRODUCTION TO ASSEMBLY LANGUAGE

Translators can be roughly divided into two groups, depending on the relation between the source language and the target language. When the source language is essentially a symbolic representation for a numerical machine language, the translator is called an **assembler** and the source language is called an **assembly language.** When the source language is a problem-oriented language, such as ALGOL 68, FORTRAN, PASCAL or PL/I, and the target language is either a numerical machine language or a symbolic representation for one, the translator is called a **compiler.**

6.1.1. What is an Assembly Language?

A pure assembly language is a language in which each statement produces exactly one machine instruction. In other words, there is a one-to-one correspondence between machine instructions and statements in the assembly program. If each card in the assembly program contains one assembly statement and each machine word contains one machine instruction, then an N-card assembly program will produce an N-word machine language program.

The reason that people use assembly language, as opposed to programming in machine language, is that it is much easier to program in assembly language. The use of symbolic names and symbolic addresses instead of binary or octal ones makes an enormous difference. Most people can remember that the abbreviations for add,

subtract, multiply, and divide are ADD, SUB, MUL, and DIV, but few can remember that the machine instructions (for the PDP-11) are 24576, 57344, 28672, and 61440. The assembly language programmer need only remember the symbolic names ADD, SUB, MUL, DIV, since the assembler translates them to the machine instructions; but the machine language programmer must remember, or constantly look up, the numerical values.

The same remarks apply to addresses. The assembly language programmer can give symbolic names to memory locations and have the assembly worry about supplying the correct numerical values. The machine language programmer must always work with the numerical values of the addresses. As a result, no one programs in machine language today although people did so years ago, before assemblers had been invented.

Assembly languages have another property, besides the one-to-one mapping of assembly language statements onto machine instructions, that distinguishes them from problem-oriented languages. The assembly programmer has access to all the features and instructions available on the target machine. The problem-oriented language programmer does not. For example, if the target machine has an overflow indicator, an assembly language program can test it; however, a FORTRAN program cannot directly test it. If there are switches on the operator console, an assembly language program can read their status. Such a program can execute every instruction in the instruction set of the target machine, but the problem-oriented language program cannot. In short, everything that can be done in machine language can be done in assembly language, but many instructions, registers, and similar features are not available for the problem-oriented language programmer to use. Languages for system programming are often a cross between these two types, with the syntax of a problem-oriented language but the access to the machine of an assembly language.

6.1.2. Format of an Assembly Language Statement

Although the structure of an assembly language statement closely mirrors the structure of the machine instruction that it represents, assembly languages for different machines and different levels have sufficient resemblance to one another to allow a discussion of assembly language in general. Figure 6-1 shows fragments of assembly language programs for the 370, Cyber 70, and PDP-11, all of which perform the computation N = I + J − K. In all three examples, the statements above the dots perform the calculation. The statements below the dots are commands to the assembler to reserve memory for the variables I, J, K, and N and are not symbolic representations of machine instructions. Statements that are commands to the assembler are called **pseudoinstructions.**

Assembly language statements have four parts: label field, operation field, operands field, and comments field. Labels, which are used to provide symbolic names for memory addresses, are needed on executable statements so that the statements can be jumped to. They are also needed on memory-allocation pseudoinstructions (e.g., DC, DATA, and .WORD), to permit the data stored there to be accessible by sym-

```
Label field   Operation field   Operands field   Comments field

FORMUL        L                1,I              LOAD I INTO REGISTER 1
              A                1,J              ADD J TO REGISTER 1
              A                1,K              ADD K TO REGISTER 1
              ST               1,N              STORE SUM IN N
                  .
                  .
                  .
I             DC               F'2'             RESERVE A WORD INITIALIZED TO 2
J             DC               F'3'             RESERVE A WORD INITIALIZED TO 3
K             DC               F'4'             RESERVE A WORD INITIALIZED TO 4
N             DC               F'0'             RESERVE A WORD INITIALIZED TO 0
```

(a)

```
FORMUL        SA1              I                LOAD I INTO X1
              SA2              J                LOAD J INTO X2
              SA3              K                LOAD K INTO X3
              IX6              X1+X2            X6 = I + J
              IX6              X6+X3            X6 = I + J + K
              SA6              N                STORE SUM IN N
                  .
                  .
                  .
I             DATA             2                RESERVE A WORD INITIALIZED TO 2
J             DATA             3                RESERVE A WORD INITIALIZED TO 3
K             DATA             4                RESERVE A WORD INITIALIZED TO 4
N             DATA             0                RESERVE A WORD INITIALIZED TO 0
```

(b)

```
FORMUL:       MOV              I,R1        ;LOAD I INTO REGISTER 1
              ADD              J,R1        ;ADD J TO REGISTER 1
              ADD              K,R1        ;ADD K TO REGISTER 1
              MOV              R1,N        ;STORE SUM IN N
                  .
                  .
                  .
I:            .WORD            2           ;RESERVE A WORD INITIALIZED TO 2
J:            .WORD            3           ;RESERVE A WORD INITIALIZED TO 3
K:            .WORD            4           ;RESERVE A WORD INITIALIZED TO 4
N:            .WORD            0           ;RESERVE A WORD INITIALIZED TO 0
```

(c)

Fig. 6-1. Computation of the formula N = I + J + K.
(a) IBM 370. (b) Cyber 70. (c) PDP-11.

bolic name. If a statement is labeled, the label (usually) begins in column 1 or 2. Figure 6-1 shows five labels: FORMUL, I, J, K, and N.

It is an unfortunate characteristic of some assemblers that labels are restricted to six or eight characters. In contrast, PL/I, ALGOL 68, PASCAL, SNOBOL4, and most other problem-oriented languages allow the use of arbitrarily long names. Long, well-chosen names make programs much more readable and understandable by other people (see Fig. 2-2 for an example of this point).

The operation field contains either a symbolic abbreviation for the opcode—if

Pseudoinstruction description	370	Cyber 70	PDP-11
Start of an assembly language program	START	IDENT	.TITLE
Reserve memory for a variable and initialize it	DC	DATA	.WORD
Reserve a block of memory without initializing	DS	BSS	.BLKW
Continue the listing on top of the next page	EJECT	EJECT	.PAGE
Equate name to value	EQU	EQU	=
Declare a label as an entry point	ENTRY	ENTRY	.GLOBL
Repeatedly assemble the following lines	REPRO	DUP	.REPT
Begin conditional assembly	AIF	IFEQ IFGT etc	.IF
Begin a macro definition	MACRO	MACRO	.MACRO
End a macro definition	MEND	ENDM	.ENDM
Begin a new program section	CSECT	USE	.CSECT

Fig. 6-2. Some common pseudoinstructions and their mnemonics on the 370, Cyber 70, and PDP-11.

the statement is a symbolic representation for a machine instruction—or a pseudoinstruction if the statement is a command to the assembler. The choice of an appropriate abbreviation is often a matter of taste, and different assembly language designers often choose different abbreviations. The 370 assembler uses "A" for integer addition. The Cyber 70 assembler uses "IX6" to indicate integer addition on X registers with the result being put in register X6. The PDP-11 assembler uses "ADD" for integer addition.

Similarly, the three assemblers have different names for the pseudoinstruction that reserves memory for variables. IBM calls it "DC" for Define Constant, CDC calls it DATA, and DEC calls it .WORD. All PDP-11 pseudoinstructions begin with a period to make them easily distinguishable. Figure 6-2 lists some common pseudoinstructions.

The operands field of an assembly language statement is used to specify the addresses and registers used as operands by the machine instruction. The operands field of an integer addition instruction tells what is to be added to what. The operands field of a jump instruction tells where to jump to. The operands field of a pseudoinstruction depends on the pseudoinstruction—for example, how much memory space to reserve, the name of the program, under what condition to assemble the following statements.

The comments field provides a place for programmers to put helpful explanations of how the program works for the benefit of other programmers who may subsequently use or modify the program. An assembly language program without such documentation is almost incomprehensible to all programmers except perhaps its author. And if a long time has passed since it was written, an assembly language program with no comments may be incomprehensible to its author as well. The comments field is

solely for the benefit of human beings and has no effect on the assembly process or the generated program.

6.1.3. Comparison of Assembly Language and PL/I

A popular myth is that programs that will be used a great deal, especially large ones, should, for efficiency, be written entirely in assembly language. Although true at one time, it is no longer. It is instructive to compare the MULTICS system with the IBM time-sharing system for the 360/67, called TSS/67. Both operating systems were started about the same time (mid-1960s) and are roughly the same size (huge). MULTICS was written, for the most part (95 percent), in a problem-oriented language, PL/I, whereas TSS/67 was written entirely in assembly language.

A large operating system like MULTICS is a stringent test for a problem-oriented language. Because an operating system must control all the i/o devices, deal with critical timing situations, handle large data bases, and do many other tasks, good performance is especially crucial. If problem-oriented languages can pass this critical test, there are few applications, indeed, where one could demonstrate the advantages to be gained by using assembly language.

The results of these two projects can be summed up nicely by pointing out that each took about the same length of time to get running. However, MULTICS was written by 50 people at an estimated cost of $10 million, whereas TSS/67 needed 300 people at an estimated cost of $50 million (Graham, 1970). The conclusion that the use of PL/I saved the MULTICS project millions of dollars is inescapable.

Studies have shown that the number of lines of debugged code a programmer can produce per month on a project over a period of several years is approximately 100 to 200 lines, independent of the programming language used (Corbató, 1969). Only on small programs can higher productivity be expected. Since one PL/I statement is equivalent to 5 to 10 assembly language statements, the productivity of a PL/I programmer will be 5 to 10 times that of an assembly language programmer.

Another strong argument against programming in assembly language is that understanding someone else's assembly language program is nearly impossible. A complete listing of MULTICS in PL/I is about 3000 pages, hardly something one digests in an evening. Nevertheless, absorbing that amount would be trivial compared to reading 30,000 pages of assembly code. Although no one ever tries to read the entire MULTICS listing, people do try to understand particular procedures, about four pages of PL/I on the average. Personnel turnover on large projects often averages 15 percent per year; consequently, after 5 years few of the original programmers are still around. If the new programmers cannot understand their predecessors' programs, the project will be in big trouble.

6.1.4. Program Tuning

Studies have shown that, in most programs, a small percentage of the total code is responsible for a large percentage of the execution time (Dardner and Heller, 1970). It is common to have 1 percent of the program be responsible for 50 percent of the

execution time and 10 percent of the program be responsible for 90 percent of the execution time. A common situation is in a compiler, where searching the symbol table can eat up more time than the rest of the compiler combined.

Assume, for example, that it requires 10 man-years to write some big compiler in a problem-oriented language and that the resulting compiler requires 100 sec to compile a certain test program. Writing the whole compiler in assembly language would require about 50 to 100 man-years, due to the lower productivity of assembler language programmers; the final program, however, would perform the test in about 33 sec, since a clever programmer can outdo a clever compiler by a factor of 3. This situation is illustrated in Fig. 6-3.

	Man-years needed to produce program	Program execution time in seconds
Assembly language	50	33
Problem oriented language	10	100
Mixed approach before tuning		
Critical 10%	1	90
Other 90%	9	10
Total	10	100
Mixed approach after tuning		
Critical 10%	6	30
Other 90%	9	10
Total	15	40

Fig. 6-3. Comparison of assembly language and problem-oriented language programming, with and without tuning.

Another approach to producing software is based on the empirical observation that, for most programs, a small percentage of the total program is responsible for nearly all the execution time. Using this approach, the entire program is first written in a problem-oriented language. Then a series of measurements is performed to determine which parts of the program account for most of the execution time. Such measurements would normally include using the system clock to compute the amount of time spent in each procedure, keeping track of the number of times each loop is executed, and similar steps.

As an example, let us assume that 10 percent of the total program accounts for 90 percent of the execution time. This means that for a 100-sec job, 90 sec are spent in this critical 10 percent and 10 sec are spent in the remaining 90 percent of the program. The critical 10 percent can now be improved by rewriting it in assembly language. This process is called **tuning** and is illustrated in Fig. 6-3. Here an additional 5 man-years are needed to rewrite the critical procedures, but their execution time is reduced from 90 to 30 sec.

It is instructive to compare the mixed problem-oriented language/assembly language approach with the pure assembly language version (Fig. 6-3). The latter is about 20 percent faster, but at more than triple the price. Furthermore, the advantage of the mixed approach is actually greater than indicated, for recoding an already debugged problem-oriented language procedure in assembly code is, in fact, much easier than writing the same assembly code procedure from scratch. In other words, the estimate of 5 man-years to rewrite the critical procedures is extremely conservative. If this recoding actually took only 1 man-year, the cost ratio between the mixed approach and the pure assembly language approach would be more than 4 to 1 in favor of the mixed approach.

A programmer who uses a problem-oriented language is not immersed in moving bits around, and sometimes he obtains insights into the problem that allow real improvements in performance. This situation occurs much less often with assembly language programmers, who are usually trying to juggle instructions to save a few microseconds. Graham (1970) reports a PL/I procedure in MULTICS that was rewritten in 3 months with the new version 26 times smaller and 50 times faster than the original, as well as another that became 20 times smaller and 40 times faster with 2 months of work.

Corbató (1969) describes a PL/I drum management procedure that was reduced from 50,000 to 10,000 words of compiled code in less than a month and an i/o controller that shrunk from 65,000 to 30,000 words of compiled code, with an improvement in speed of a factor of 8 in four months. The point here is that because problem-oriented-language programmers have a more global view of what they are doing, they are far more likely to obtain insights leading to totally different and vastly better algorithms.

Despite all these disadvantages, with almost no compensating advantages, some people still program in assembly language. The reason why is essentially a hangover from the days when machine time was so expensive that wasting months or even years of programmer time to save a few hours or days of machine time was understandable. It has been estimated that 80 percent of total computer costs at present are people costs and 20 percent are machine costs; so having a team of $20,000-a-year (including overhead) programmers work for 6 months in order to save 10 hours of machine time that costs $500 per hour simply does not make sense.

After this introduction, you may wonder: "Why bother studying assemblers at all when assembly language programming has so many serious drawbacks?" There are, in fact, three reasons. First, since the success or failure of a large project may depend on being able to squeeze a factor of 5 or 10 improvement in performance out of some critical procedure, it is important to be able to write good assembly language code when it is absolutely necessary. Second, some of the techniques used in an assembler (e.g., table lookup) are essentially the same techniques used in a compiler. Consequently, this chapter also has relevance to compiler writing, an important subject in computer science. Third, a compiler must either produce output used by an assembler or perform the assembly process itself.

6.2. THE ASSEMBLY PROCESS

In the following sections we will describe how an assembler works.

6.2.1. Two-Pass Assemblers

Because an assembly language program consists of a series of one-line statements, it might seem natural to have an assembler that read one statement, translated it to machine language, and then output the generated machine language onto a file. The process would then be repeated until the whole program had been translated. Unfortunately, this method does not work.

Consider the situation where the first statement is a jump to L. The assembler cannot assemble this instruction until it knows the address of statement L. Statement L may be near the end of the program, making it impossible for the assembler to find the address without first reading almost the entire program. This difficulty is called the forward-reference problem because a symbol, L, has been used before it has been defined; that is, a reference has been made to a symbol whose definition will only occur later.

Forward references can be handled in two ways. First, the assembler may, in fact, read the source program two times. Each reading of the source program is called a **pass,** and any translator that reads the input program two times is called a **two-pass translator.** On pass 1 of a two-pass assembler, the definitions of symbols, including statement labels, are collected and stored in a table. By the time the second pass begins, the values of all symbols are known; thus there is no longer a forward-reference problem, and each statement can be read, assembled, and output.

The second method consists of trying to do the assembly in one pass anyway. Whenever a statement is encountered that cannot be assembled because it contains a forward reference, no output is generated; instead an entry is made in a table, indicating that the statement with the forward reference has not yet been assembled. At the end of the assembly, all symbols will have been defined, so all statements in the not-yet-assembled table can be assembled.

The latter method generates output in a different order than the two-pass method. If the assembly is followed by loading, the loader can put the pieces of output back in the right order. Hence this objection is not a serious one. The one-pass assembler does have the problem that if many statements contain forward references, the table containing the statements not yet assembled may become too large to fit in memory. The assembler could write the table out into secondary memory and then read it back later, however.

One-pass assemblers tend to be more complicated than two-pass assemblers. For example, assembly language programmers expect a program listing that contains the source statements listed in the order they appear in the program, along with the machine language translation of each. The one-pass assembler cannot print the machine translation until it has actually done it. Consequently, it must have either a

second pass to print the listing or another internal table where pieces of the listing are kept until they can be printed. Most assemblers are two pass.

6.2.2 Pass One

The principal function of pass one is to build up a table called the **symbol table,** containing the values of all symbols. A symbol is either a label or a value that is assigned a symbolic name by means of an explicit pseudoinstruction, such as

```
BUFSIZE   EQU   100
```

In assigning a value to a symbol in the label field of an instruction, the assembler must know what address that instruction will have during execution of the program. In order to keep track of the execution-time address of the instruction being assembled, the assembler maintains a variable during assembly, called the **instruction location counter** or ILC. This variable is set to 0 at the beginning of pass one and incremented by the instruction length for each instruction processed, as shown in Fig. 6-4. The instruction length is determined by looking up the opcode in the table and, on some computers, such as the PDP-11, making a preliminary analysis of the operands as well.

Label field	Operation field	Operands field	Comments	Instruction length in bytes	ILC value (decimal) just before the statement is read
	⋮				
SUZANNE	L	2,I	R2 = I	4	100
	L	4,J	R4 = J	4	104
	L	6,K	R6 = K	4	108
MARIA	MR	2,2	R2 = I * I	2	112
	MR	4,4	R4 = J * J	2	114
	MR	6,6	R6 = K * K	2	116
MARILYN	AR	2,4	R2 = I * I + J * J	2	118
	AR	2,6	R2 = I * I + J * J + K * K	2	120
CAROL	ST	2,N	N = I * I + J * J + K * K	4	122
	B	DONE	EXIT TO DONE	4	126
	⋮				

Fig. 6-4. The instruction location counter (ILC) keeps track of the address where the instruction will be loaded in memory. In this example, the statements prior to SUZANNE occupy 100 bytes.

Some assemblers allow programmers to write immediate instructions even though no corresponding target language instruction exists. Such "pseudo-immediate" instructions are handled as follows. The assembler allocates memory for the immediate operand at the end of the program and generates an instruction that references it. For instance, in 370 assembly language, a programmer can write

```
L    1,=F'2'
```

to load register 1 with a Full word constant 2. In this way, the programmer avoids explicitly writing a DC pseudoinstruction so as to allocate a word initialized to 2. Constants for which the assembler automatically reserves memory are called **literals**. In addition to saving the programmer a little writing, literals improve the readability of a program by making the value of the constant apparent in the source statement. Pass one of the assembler must build a table of all literals used in the program.

Pass one uses at least three tables: the symbol table, the opcode table, and the literal table. The symbol table has an entry for each symbol, as shown in Fig. 6-5.

Symbol	Value	Other information
SUZANNE	100	
MARIA	112	
MARILYN	118	
CAROL	122	

Fig. 6-5. A symbol table for the program of Fig. 6-4.

Symbols are defined either by using them as labels or by explicit definition (e.g., EQU on the 370 and Cyber 70). Each symbol table entry contains the symbol itself (or a pointer to it), its numerical value, and sometimes other information. This additional information may include

1. length of data field associated with symbol.
2. relocation bit (Does the symbol change value if the program is loaded at a different address than the assembler assumed?).
3. whether or not the symbol is to be accessible outside the procedure.

The opcode table contains one entry for each symbolic opcode (mnemonic) in the assembly language. Figure 6-6 shows part of an opcode table. Each entry contains the symbolic opcode, its numerical value, and a class number that separates the opcodes into groups, depending on the number and type of operands. For example,

Symbolic opcode	Opcode in hexadecimal	Instruction length in bytes	Instruction class
A	5A	4	1
AH	4A	4	1
MR	1C	2	7
CR	19	2	7
MVC	D2	6	4
L	58	4	1
B	47	4	9

Fig. 6-6. Part of the opcode table for an assembler for the 370.

the 370 RR, RX, RS, SI, and SS instructions have different formats for their operands, requiring a different procedure for analyzing each one. On some machines, such as the 370 and Cyber 70, the length of the instruction is uniquely determined by the opcode, but on others, such as the PDP-11, it is not. The PDP-11 MOV instruction takes two operands, either of which can be a register or memory location, yielding three different lengths:

```
MOV  R0,R1    register to register    (2 bytes)
MOV  R0,X     register to memory      (4 bytes)
MOV  X,R0     memory to register      (4 bytes)
MOV  X,Y      memory to memory        (6 bytes)
```

The third table is the literal table. For each literal, one entry is made in the table. An entry contains the character string representing the literal, the value of the literal in binary, and the address where the literal is stored. The literals will eventually be stored in memory in locations following the object program. This means that the address where a particular literal is to be stored will only be known when the program length is known—that is, after pass one is finished.

If a literal is used several times in the program, it is desirable, nevertheless, to allocate only one copy of it in the object program. This can be done in two ways. If the literal table is searched before a literal is entered—to see if it is already there—duplicate entries can be prevented. Alternatively, all literals are entered; and after pass one, the literal table is sorted and duplicates removed.

Figure 6-7 shows a procedure that could serve as a basis for pass one of an assembler. The style of programming is noteworthy in itself. The procedure names have been chosen so as to give a good indication of what the procedures do. Most important, Fig. 6-7 represents an outline of pass one, which, although not complete, forms a good starting point. It is short enough to be easily understood, and it makes clear what the next step must be—namely, to write the procedures used in it.

Some of these procedures will be quite short, such as CHECK_FOR_LABEL, which returns the label as a character string if there is one and a blank string if there is not. Other procedures, such as TYPE_1 and TYPE_2, may be longer and may call other procedures. In general, the number of classes will not be two, of course, but will depend on the language being assembled.

Structuring programs in this way has other advantages in addition to ease of programming. If the assembler is being written by a group of people, the various procedures can be parceled out among the programmers. All the (nasty) details of obtaining input are hidden away inside READ_NEXT_STATEMENT. If they should change—for example, due to an operating system change—only one subsidiary procedure is affected, and no changes are needed to the PASS_1 procedure itself.

In some assemblers, after a statement has been read in, it is stored in a table. If the table should fill up, it must be written out into secondary memory. It may fill up several times and be written out several times. If the program being assembled is short enough to fit in the table, pass two can get its input directly from the table, thus saving the time needed to write it out and read it back in again.

When the END pseudoinstruction is read, pass one is over. The symbol table and

```
PASS_1:  PROCEDURE;

    /* THIS IS AN OUTLINE OF PASS ONE OF A SIMPLE ASSEMBLER */

    DECLARE (LOCATION_COUNTER,CLASS,MORE_INPUT,LENGTH,COM,VALUE) FIXED,
            (LITERAL,LABEL) CHARACTER(8), OPCODE CHARACTER(6),
            LINE CHARACTER(80);

    LOCATION_COUNTER = 0;
    MORE_INPUT = 1;
    CALL INITIALIZE_TABLES;

    DO WHILE (MORE_INPUT = 1);
       /* THIS LOOP IS EXECUTED ONCE PER CARD READ */
       CALL READ_NEXT_STATEMENT(LINE);
       CALL SAVE_LINE_FOR_PASS2(LINE);
       CALL IS_THIS_A_COMMENT(LINE,COM);

       /* COM IS SET TO 0 FOR COMMENT LINES AND 1 OTHERWISE */
       IF COM = 1
          THEN DO; /* NO COMMENT */
                   CALL CHECK_FOR_LABEL(LINE,LABEL);
                   IF LABEL ¬= ' '
                     THEN CALL ENTER_NEW_SYMBOL(LABEL,LOCATION_COUNTER);
                   CALL CHECK_FOR_LITERAL(LINE,LITERAL);
                   IF LITERAL ¬= ' '
                      THEN CALL ENTER_LITERAL(LITERAL);

                   /* NOW DETERMINE THE NATURE OF THE OPCODE. THE
                      OPCODES ARE DIVIDED INTO CLASSES, WITH ALL MEMBERS
                      OF A GIVEN CLASS HAVING THE SAME TYPE AND NUMBER OF
                      OPERANDS. IF OPCODE IS NOT FOUND, CLASS = -1 */
                   CALL EXTRACT_OPCODE(LINE,OPCODE);
                   CALL SEARCH_OPCODE_TABLE(OPCODE,CLASS,VALUE);
                   IF CLASS = -1
                      THEN CALL TRY_PSEUDO_INSTR(OPCODE,CLASS,VALUE);
                   LENGTH = 0;
                   IF CLASS < 0 THEN CALL ILLEGAL_OPCODE;
                   IF CLASS = 1 THEN CALL TYPE_1(LINE,LENGTH);
                   IF CLASS = 2 THEN CALL TYPE_2(LINE,LENGTH);

                   /* TEST FOR OTHER CLASSES HERE */

                   LOCATION_COUNTER = LOCATION_COUNTER + LENGTH;
                   IF CLASS = 99 /* END STATEMENT */
                      THEN DO;
                              MORE_INPUT = 0;
                              CALL REWIND_PASS2_INPUT;
                              CALL SORT_LITERAL_TABLE;
                              CALL REMOVE_REDUNDANT_LITERALS;
                           END;
               END;
       END;
    END PASS_1;
```

Fig. 6-7. Pass one of a simple assembler.

literal tables can be sorted at this point if needed. The sorted literal table can be checked for duplicate entries, which can be removed. The literal table is then compacted, the values are computed, and an address is assigned to each literal, beginning at the first word following the program being assembled.

6.2.3 Pass Two

The function of pass two is to generate the object program and print the assembly listing. In addition, pass two must output certain information needed by the linker for linking up procedures assembled at different times. Figure 6-8 shows a procedure for pass two.

```
PASS_2: PROCEDURE;

    /* THIS IS AN OUTLINE OF PASS TWO OF A SIMPLE ASSEMBLER */

    DECLARE (OPERANDS(1:3),CODE,CLASS,VALUE,MORE_INPUT,LOCATION_LENGTH,
         COM) FIXED, LINE CHARACTER(80),OPCODE CHARACTER(6);

    MORE_INPUT = 1;
    LOCATION_COUNTER = 0;

    DO WHILE(MORE_INPUT);
       CALL GET_NEXT_STATEMENT(LINE);
       CALL IS_THIS_A_COMMENT(LINE,COM);

       /* COM IS SET TO 0 FOR COMMENT LINES, AND 1 OTHERWISE */
       IF COM = 1
         THEN DO;
                  CALL EXTRACT_OPCODE(LINE,OPCODE);
                  CALL SEARCH_OPCODE_TABLE(OPCODE,CLASS,VALUE);
                  IF CLASS = -1
                     THEN CALL TRY_PSEUDO_INSTR(OPCODE,CLASS,VALUE);
                  LENGTH = 0;
                  IF CLASS < 0 THEN CALL ILLEGAL_OPCODE_2;
                  IF CLASS = 1 THEN CALL EVAL_1(LINE,OPERANDS,LENGTH);
                  IF CLASS = 2 THEN CALL EVAL_2(LINE,OPERANDS,LENGTH);

                  /* TEST FOR THE OTHER CLASSES HERE */

                  CALL ASSEMBLE_PIECES(CODE,CLASS,VALUE,OPERANDS);
                  CALL OUTPUT_INSTRUCTION(CODE);
                  LOCATION_COUNTER= LOCATION_COUNTER+LENGTH;
              END;

      CALL PRINT_LISTING(LINE,CODE,COM,LOCATION_COUNTER);
      IF CLASS = 99   /* TEST FOR END */
         THEN DO;
                  MORE_INPUT = 0;
                  CALL FINISH_UP;
              END;
    END;

END PASS_2;
```

Fig. 6-8. Pass two of a simple assembler.

The procedure for each class knows how many operands that class may have and calls the EVALUATE_EXPRESSION procedure (not shown) the appropriate number of times. The EVALUATE_EXPRESSION procedure must convert the symbolic expression to a binary number. It must first find the values of the symbols and the addresses of the literals from the respective tables. Once the numerical values are known, the expression can be evaluated. There are numerous techniques for evaluating arithmetic expressions. One method (described in Section 3.3.7) is to convert the expression to reverse Polish and evaluate it by using a stack.

Once the numerical value of the opcode and the values of the operands are all known, the complete instruction can be assembled. The assembled instruction is then placed in an output buffer that is written into secondary memory or punched on cards when the buffer becomes full.

The original source statement and the object code generated from it (in octal or hexadecimal) are either printed or put into a buffer for later printing. After the ILC has been adjusted, the next statement is fetched.

Up until now it has been assumed that the source program does not contain any errors. Anyone who has ever written a program, in any language, knows how realistic that assumption is. Some of the common errors are as follows:

1. A symbol has been used but not defined.
2. A symbol has been defined more than once.
3. The name in the opcode field is not a legal opcode.
4. An opcode is not supplied with enough operands.
5. An opcode is supplied with too many operands.
6. An octal number contains an 8 or 9.
7. A register expression has been used where that is not allowed (e.g., a jump to a register).
8. END statement missing.

Programmers are most ingenious at thinking up new kinds of errors to make. Undefined symbol errors are frequently caused by typing errors, and a very clever assembler could try to figure out which of the defined symbols most resembles the undefined one and use that instead. There is little that can be done about correcting most other errors. The best thing for the assembler to do with an errant statement is to assemble a HALT instruction in its place.

If an incorrect statement is treated as a HALT, the program can be loaded and executed anyway. If the program never executes the bad statement, no harm will be done. If it does, the program will halt at that point, but the programmer may have been able to find a few bugs in the statements that did execute. Not allowing the program to execute because a single character was punched wrong would seem to violate the U.S. Constitution's prohibition against cruel and unusual punishment.

6.3. SEARCHING AND SORTING

Assemblers and compilers need the ability to associate names with values. The symbol table in the assembler above provides a mapping of symbolic names onto numerical values. What is really needed is an **associative memory**. An associative memory has two part entries, as shown in Fig. 6-9. In order to look up the value associated with a name, the name is put into the MAR. The associative memory hardware then simultaneously compares all keys to the MAR and puts the value associated with the matching key in the MBR. If such a memory were available at a low enough cost, the symbol-table searching could be done directly in the hardware. Until the time comes when associative memories are cheap enough to eliminate searching, it will be necessary to simulate them in software. The next few sections explain some algorithms for simulating associative memories.

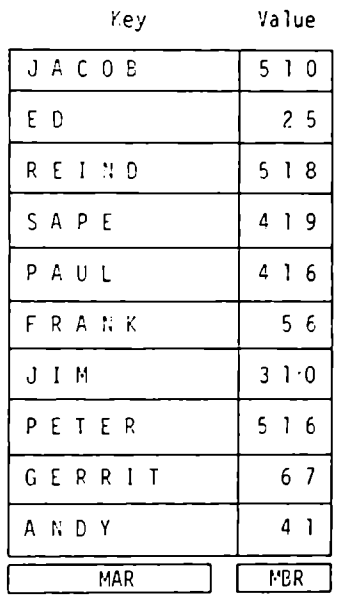


Fig. 6-9. An associative memory. Each entry has two parts, a key and a value. When a string is put into the MAR, all the keys are simultaneously compared to it and the corresponding value is put into the MBR.

6.3.1. Searching

Searching is the process of looking something up in a table. A search procedure takes as its input the name, or **search key**, it is to find in the table and delivers as its output the value associated with that key. Because the value may consist of several fields of information, search procedures commonly deliver an index into the table telling where the value is located, instead of the value itself.

Since every statement in an assembly language program contains an opcode that must be looked up in the opcode table and since most statements contain one or more symbols as operands as well, it is important that searching be performed as quickly as possible. It is not unusual to find assemblers and compilers that spend more than half their time searching various tables. The linear search described in the next section is

the easiest to program, but it is not nearly as fast as the binary search described in the section following it.

6.3.2. Linear Searching

A procedure for linear searching needs three parameters—the table to be searched, containing a series of ordered pairs (key, value), the search key, and the number of entries in the table. For simplicity, assume that all tables to be searched consist of one-dimensional arrays with elements numbered consecutively from 1 to N. The array may be larger than N—that is, not full.

The simplest search algorithm, illustrated in Fig. 6-10, is the linear search. It compares the key of the first entry to the key being sought. If they match, the search is over and the value of the search procedure is 1, corresponding to the fact that the first entry matched the search key. If they do not match, the search key is compared to the second key in the table. If the search key matches the second key, the search is over; otherwise the third key is tried, and so on. Eventually, either the search key is found or every entry in the table has been tried with no success, meaning that the search key is not in the table. If entry *i* matches the search key, the procedure of Fig. 6-10 returns *i* as its value. If the search key is not present in the table, the procedure returns 0 as its value.

The linear search has many advantages. First, it is easy to program. Second, the entries in the table may be in any order. The binary search (described in the next section) requires that the entries be sorted on their keys—that is, that the first entry have the smallest key, the second entry have the next smallest key, and so forth. Not having to sort the table entries is a big advantage. Third, the linear search may be used while the table is being built up. For example, some assemblers collect symbols during pass one, sort them after pass one, and use a search procedure that requires a sorted table during pass two. If, however, these assemblers must look up a symbol

```
LINEAR_SEARCH:  PROCEDURE(A,N,KEY);

    /* THIS PROCEDURE SEARCHES THE INTEGER TABLE A.  IF A MATCH IS FOUND
       FOR THE SEARCH KEY, THE INDEX OF THE MATCHING ENTRY IS RETURNED.
       IF NO MATCH IS FOUND, A 0 IS RETURNED. */

    DECLARE (A(1:1000),N,KEY) FIXED BINARY;
    DECLARE I FIXED BINARY;

    DO I = 1 TO N;

       /* DOES THIS ENTRY MATCH THE KEY? */
       IF A(I) = KEY THEN RETURN(I);
    END;

    RETURN(0);

END LINEAR_SEARCH;
```

Fig. 6-10. Linear search.

during pass one, the table is not yet sorted, so the pass-two search procedure cannot be used. Either the table must be sorted immediately or a linear search procedure must be used. If it is necessary to look up several symbols on pass one, the alternative of sorting the table several times may be costly.

The linear search procedure varies widely in performance. The best case is when the first entry matches the search key, requiring only one comparison. The worst case is when the last entry matches the search key, requiring N comparisons. The average case requires $(N + 1)/2$ comparisons. If N is large, this is approximately $0.5 \times N$. The average search time is proportional to the number of items in the table. If the table size is doubled, the search time is doubled.

6.3.3. Binary Searching

A binary (or logarithmic) search is a method that requires a sorted table and that takes advantage of that sorting to speed up the search enormously. In performing a binary search on a table whose first entry is in position M of the array and whose last entry is in position N of the array (in the previous example, M was always 1), the entry closest to the middle is compared to the search key. If they are equal, the search is over. If the search key is larger than the middle key, the key being sought must be in the second half of the table. If the search key is smaller than the middle key, the key being sought is in the first half of the table. If the key is not located, at least the half containing the key is now known, and the binary search can be used on that half.

If the keys are integers, the relations "less than" and "greater than" are easy to compute. If the keys are character strings, the comparisons refer to alphabetical order. A string s1 is less than a string s2 if and only if s1 comes before s2 alphabetically. If the strings contain characters other than letters—for example, digits, spaces, commas, periods—then some definition of alphabetical must be adopted to resolve questions like "is $X < 2$?" or "is THEDOG less than THE DOG?" or "is $B > BBB$?" As far as the binary search is concerned, it does not matter what conventions are adopted as long as they are used consistently.

Figure 6-11 illustrates a binary search for secretarybird. The middle entry $(1 + 16)/2$ is 8.5, so entry 8, marmot, is compared to the search key, secretarybird. Marmot is less than secretarybird (because m comes before s in the alphabet); thus the search for secretarybird continues, using the second half of the table—namely, the part between otter and wren inclusive. Otter is entry 9 and wren is entry 16, so the middle entry is $(9 + 16)/2 = 12.5$. Rudderfish and secretarybird are compared—alphabetically, of course. Secretarybird is the larger; thus the lower part of the otter-wren section of the table is used on step 3—namely, entries 13 to 16. The middle entry is $(13 + 16)/2 = 14.5$; therefore squid is compared to secretarybird. Squid is larger, so the next step uses that portion of the secretarybird-wren table below squid. It so happens that secretarybird is the only entry; consequently, it is found on step 4.

Figure 6-12 gives a procedure for the binary search algorithm. First, a test is made to see if LO = HI. Thus the (sub) table being searched has only one entry, and the one entry is compared to the search key. If the (sub) table has more than one entry,

Entry	Key	Value
1	Aardvark	10
2	Baboon	7
3	Camel	4
4	Dinosaur	7
5	Guinea pig	4
6	Kangaroo	4
7	Louse	4
8	Marmot	7
9	Otter	5
10	Polliwog	11
11	Python	6
12	Rudderfish	13
13	Secretarybird	17
14	Squid	11
15	Stork	8
16	Wren	14

```
Step 1:
Aardvark to wren (1-16),
middle entry is marmot (8)

Step 2:
Otter to wren (9-16),
middle entry is rudderfish (12)

Step 3:
Secretarybird to wren (13-16)
middle entry is squid (14)

Step 4:
Secretarybird to secretarybird (13-13)
middle entry is secretarybird (13)
```

Fig. 6-11. An example of a binary search for secretarybird.

```
BINARY_SEARCH: PROCEDURE(A,LO,HI,KEY);

    /* THIS PROCEDURE USES THE BINARY SEARCH ALGORITHM TO LOOK UP THE
       SEARCH KEY IN A SORTED TABLE, A. ONLY ENTRIES BETWEEN A(LO) AND
       A(HI) INCLUSIVE ARE CHECKED. THE TABLE WILL CONTAIN ONLY
       INTEGERS, FOR SIMPLICITY.  IF THE SEARCH IS SUCCESSFUL, THE INDEX
       OF THE ENTRY LOCATED IS RETURNED, OTHERWISE 0 IS RETURNED. */

    DECLARE (A(1:1000), LO,HI,KEY) FIXED BINARY;
    DECLARE MIDDLE FIXED BINARY;

    /* CHECK TO SEE IF THE TABLE CONTAINS ONLY 1 ENTRY */
    IF LO = HI
       THEN DO;
              IF KEY = A(LO) THEN RETURN (LO);  ELSE RETURN(0);
            END;
       ELSE DO;
              MIDDLE = (LO+HI)/2;  /* DIVIDE TABLE INTO 2 PARTS */
              IF KEY = A(MIDDLE)
                 THEN RETURN (MIDDLE);

                 /* CALL BINARY_SEARCH TO SEARCH 1 OF THE 2 PARTS */
                 ELSE IF KEY < A(MIDDLE)
                        THEN RETURN(BINARY_SEARCH(A,LO,MIDDLE,KEY));
                        ELSE RETURN(BINARY_SEARCH(A,MIDDLE+1,HI,KEY));
            END;

END BINARY_SEARCH;
```

Fig. 6-12. Binary search.

the index of the middle entry is stored in the integer variable MIDDLE. If the search key is smaller than the middle entry, the upper part of the table—namely, those elements between LO and MIDDLE − 1 inclusive—must be searched. The easiest way is to call BINARY_SEARCH recursively. If the search key is greater than the middle

entry, the portion of the table between MIDDLE + 1 and HI must be searched. Again, a recursive call to BINARY_SEARCH is the easiest way.

Since binary searching reduces the table size by slightly more than a factor of 2 on each successive call, a table consisting of 2^N entries will require N comparisons at most. On the average, a binary search will make about $\log_2 N$ comparisons. For a table of 1000 entries, this is about 10 comparisons, versus 500 for a linear search, quite an improvement.

Estimated entry searching is an interesting variation on the binary search. It is the method that people generally use (unconsciously) when looking something up in a sorted list. Consider how the average man on the street would look up the telephone number of Zibignieuw Zworykin in the Manhattan telephone book. Would he open the book in the middle? Of course not. He would open it to one of the last few pages. Based on the search key, Zworykin, he makes an estimate of where the entry is and looks there. If the page that he opens to begins with Yvette Yak, he repeats the estimation on that part of the book following Yak.

Making good estimates requires using some knowledge about the data. For example, if the keys are known to be integers uniformly distributed between 0 and R and the table has N entries, a simple estimate of the position of a key, K, is NK/R. For many applications, a good estimate can be made for the first few tries; but as the table gets smaller, statistical fluctuations in the distribution may reduce estimates to random guesses. A possible search strategy is to use an estimated entry search for the first few attempts, and then switch to a binary search for the rest. The average number of comparisons depends on how good the estimates are.

6.3.4. Sorting

The binary search algorithm requires that the names in the symbol table be sorted in alphabetical order. Literally hundreds of sorting algorithms are known, and the list is growing constantly. No one algorithm is best in all situations. Some algorithms are fast but require a large amount of working space. Others are only appropriate when there is a very large amount of data—for example, 10 tapes full of customer names and addresses.

One of the simplest possible sorting algorithms is shown in Fig. 6-13. The sort begins by examining the first two entries. If the first one is larger, they are interchanged. Then the second and third entries are compared and interchanged if they are out of order. Succeeding adjacent entries are also compared until the end of the table is reached. At this point, the largest key will be in position N. This process, called a pass, is then repeated on the first N − 1 entries, the result of which is to place the second largest entry in table position N − 1. The third pass is made on entries 1 to N − 2 and so forth, until the entire table has been sorted. This sort requires N(N − 1)/2 comparisons, so doubling the table size will more than quadruple the sorting time. For large tables, a better algorithm is needed.

A fast, simple sorting algorithm that requires no extra storage is Shell sort. Shell sort is named after its creator, D. L. Shell (1959) and has nothing to do with shells.

```
INTERCHANGE_SORT:  PROCEDURE(A,N):

    /* THIS PROCEDURE SORTS THE INTEGER TABLE A USING A SIMPLE SORTING
       ALGORITHM.  PAIRS OF NUMBERS ARE COMPARED AND INTERCHANGED IF
       THEY ARE IN THE WRONG ORDER. */

    DECLARE (A(1:1000),N) FIXED BINARY;
    DECLARE (I,J,TEMPORARY) FIXED BINARY;

    DO I = 1 TO N-1;
       DO J = 1 TO I;
          IF A(J) > A(J+1)
             THEN DO;
                     /* INTERCHANGE A(J) AND A(J+1) */
                     TEMPORARY = A(J);
                     A(J) = A(J+1);
                     A(J+1) = TEMPORARY;
                  END;
       END;
    END;

END INTERCHANGE_SORT;
```

Fig. 6-13. Interchange sort.

It consists of a number of comparisons of two keys. If the larger comes before the smaller in the table, the entries are interchanged. In this way, entries with low keys are moved toward the beginning of the table and entries with high keys are moved toward the end of the table, thus tending to order the entries and produce a sorted table. During the first pass, entries relatively far apart are compared in order to move those with low keys that were initially near the end of the table to the beginning, and vice versa. Subsequent passes use a steadily decreasing distance between the compared entries. Figure 6-14 gives a procedure for Shell sort. Loeser (1974) compares the performance of Shell sort with more complicated (and faster) sorting algorithms.

6.3.5. Hash Coding

The function of a symbol table is to simulate an associative memory. The previous sections have examined one technique for doing so: a binary search of a sorted table. There exists another method for simulating an associative memory that is faster, more direct, and that does not require a sorted table. It is called **hash coding**.

Each symbol table entry consists of a name and a value associated with it. Hash coding consists of taking the name and, from it, generating an integer that is the table position at which the name and value are table stored. The function that converts a name into an integer from 1 to the table size is called the **hashing function**. An ideal hashing function would generate a different table position for each name, so that, given a name, its table position could be calculated directly and no table searching would be needed. The same hashing function used to enter names in the table is used to look them up.

```
SHELL_SORT:  PROCEDURE(A,N);

    /* THIS PROCEDURE SORTS THE INTEGER ARRAY A.  THERE ARE N NUMBERS
       TO BE SORTED.  LIKE INTERCHANGE SORT, PAIRS OF ENTRIES ARE COMPARED
       AND INTERCHANGED IF THEY ARE IN THE WRONG ORDER.  THE DISTANCE
       BETWEEN ENTRIES COMPARED DECREASES ON SUCCESSIVE PASSES.  FOR
       EXAMPLE, IF N = 1000, THERE WILL BE 9 PASSES, WITH A SPACING OF
       511, 255, 127, 63, 31, 15, 7, 3, AND 1 RESPECTIVELY.  THE ENTRIES
       COMPARED ARE A(I) AND A(I+DIST).  AFTER AN EXCHANGE, AN ATTEMPT
       IS MADE TO MOVE THE SMALLER VALUE UP EVEN MORE. */

    DECLARE (A(1:1000),N) FIXED BINARY;
    DECLARE (DIST,I,K,TEMPORARY) FIXED BINARY;

    /* DETERMINE THE INITIAL VALUE OF DIST BY FINDING THE LARGEST POWER OF
       2 LESS THAN N, AND SUBTRACTING 1 FROM IT.  THE FINAL STEP IN THIS
       CALCULATION IS PERFORMED INSIDE THE MAIN SORTING LOOP.  */
    DIST = 4;
    DO WHILE (DIST > N);  DIST = DIST + DIST; END;
    DIST = DIST - 1;

    /* MAIN SORTING LOOP.  THE OUTER LOOP IS EXECUTED ONCE PER PASS. */
    DO WHILE (DIST > 1);
       DIST = DIST/2;
       DO K = 1 TO N-DIST;

          /* THIS STATEMENT IS THE COMPARISON.  IT ALSO CONTROLS MOVING
             VALUES UPWARD AFTER AN EXCHANGE. */
          DO I = K BY -DIST TO 1 WHILE (A(I)>A(I+DIST));

             /* THE NEXT 3 STATEMENTS PERFORM THE EXCHANGE. */
             TEMPORARY = A(I);
             A(I) = A(I+DIST);
             A(I+DIST) = TEMPORARY;
          END;
       END;
    END;
    RETURN;

END SHELL_SORT;
```

Fig. 6-14. Shell sort.

Unfortunately, no known hashing function guarantees a unique table position for every possible name. As a result, two different names may hash to the same table position. If a name to be entered in the symbol table hashes to a position already occupied, the new entry cannot be put there, and something else must be done with it.

One possibility is to generate another table position. If the second try also fails, a third position can be generated. If this method is chosen, the hashing system must be capable of generating a whole series of table positions to ensure that an empty one will be found eventually. An alternative to generating a series of table positions is to keep all the entries hashing to position N on a linked list starting in position N.

To demonstrate some of the details involved in hash coding, the names and values shown in Fig. 6-15 will be entered into a hash table with 19 positions, numbered from

Name	Value
Magenta	8
Cyan	40
Bichromatic yellow	6
Beige	22
Mauve	18
Chartreuse	34

Fig. 6-15. Symbols and values to be used in the hash coding examples.

1 to 19. A very simple hashing function will be used here to make the example clear; more sophisticated hashing functions will be considered later. To find the table position for a name, add up the values of the letters, using a = 1, b = 2, c = 3, . . . , z = 26. The sum of the letters should be taken modulo 19 (i.e., divide the sum by 19 and use the remainder). This number will be between 0 and 18 inclusive. Since the table entries are numbered 1 to 19, and not 0 to 18, 1 must be added to it to get the table position. For example, magenta is m (=13) + a (=1) + g (=7) + e (=5) + n (=14) + t (=20) + a (=1), which is a total of 61. Dividing 61 by 19, we get a remainder of 4; so the table position is 4 + 1, and magenta will be stored in table position 5.

In order to permit arbitrarily long names, we will use two tables—a hash table and a name table. The hash table will be used to store indices into the name table, as well as the values to be associated with the names. The character string representing the name will be stored in the name table. Each name will be followed by a period to indicate its end. The number of characters per address is machine dependent: in this example we will use four characters per address. Figure 6-16(a) shows how the tables look after magenta has been entered.

The next entry is cyan. Adding c (=3) + y (=25) + a (=1) + n (=14), we get 43. The remainder when 43 is divided by 19 is 5, and 5 + 1 is 6; so the pointer to cyan goes in position 6 of the hash table, as shown in Fig. 6-16(b). Figure 6-16(c) shows the situation after magenta, cyan, bichromatic yellow, and beige have been entered. The numbers 5, 1, 3, and 10 in the hash table point to the names bichromatic yellow, magenta, cyan, and beige in the name table, beginning at addresses 5, 1, 3, and 10, respectively. Now we hash mauve by adding m (=13) + a (=1) + u (=21) + v (=22) + e(=5) to get 62, which modulo 19 is 5. Adding 1 to 5, we get 6; so mauve belongs in word 6 of the hash table. Unfortunately, word 6 is already occupied by cyan. This is a collision. Even though mauve does not look at all like cyan, they have hashed to the same value.

One simple strategy for handling collisions is to begin searching consecutive entries in the hash table, starting at the entry where the name belongs, until an empty space is found and then use that. The search should be a circular one, meaning that if word 19 is tried and is occupied, then the search continues with word 1. Of course, each position in the table is only tried once. Mauve really belongs in 6, so we examine 7, discover that it is empty, and put the pointer to mauve, 12, and its value 18 there, as in Fig. 6-16(d). Finally, we hash chartreuse, which yields a 5. Word 5 in the hash table is already occupied by magenta, so we try the next one. Word 6 is also full, so

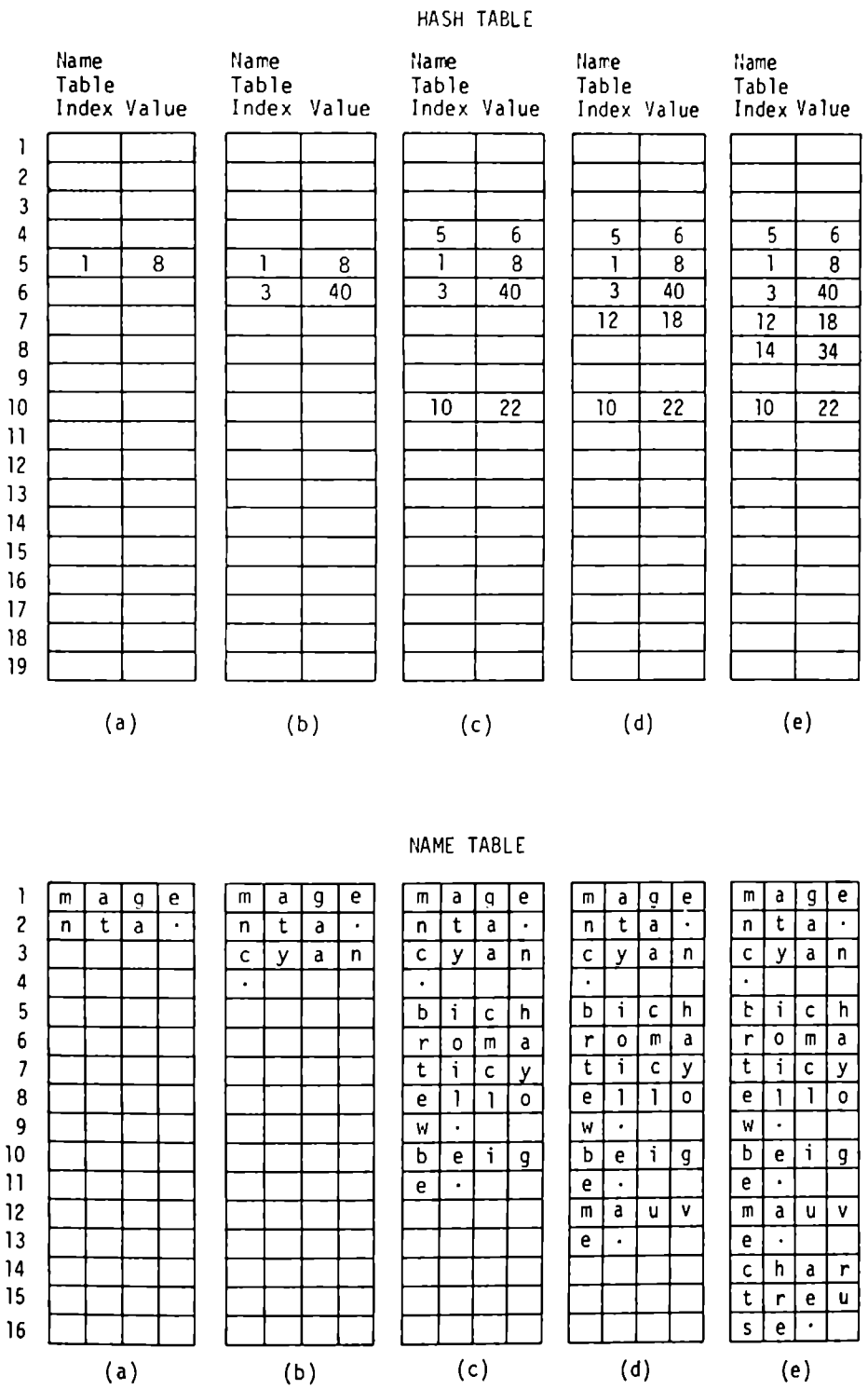


Fig. 6-16. An example of hash coding with names of arbitrary length.

we try 7, which is also full. Al last we arrive at 8, which is empty, so chartreuse goes in 8. Figure 6-16(e) shows the tables with all six names entered.

To look up a name, the hash code is generated from the name, just as it is in inserting. The index of the character string is fetched from the hash table, and the character string is compared to the search key. If they match, the search is over. If they do not, the search continues by fetching the pointer in the next hash table word (circularly again; i.e., 1 follows 19) and comparing the string to which it points with the search key. If there is still no match, successive words in the hash table are tried until one of the following three conditions is met:

1. A match is found for the search key.
2. An empty position is encountered in the hash table.
3. All 19 entries in the hash table have been tried with no success.

Conditions 2 and 3 can only occur if the search key is not in the table.

The number of attempts required to find the name being sought depends on the fraction of the table filled and the uniformity of the table positions generated by the hashing function. The first name entered can never collide with a previously entered name. If the fraction of the table filled is r, then a new name will have a probability $1 - r$ of not colliding, assuming that the hashing function generates all values with equal probability. It will take only one try to find beige in Fig. 6-16(e) but four tries to find chartreuse because of collisions. Whenever linear chains develop within the hash table (called **clustering**), as in 4, 5, 6, 7, 8, the lookup becomes a linear search. In fact, the linear search is really a special case of hash coding in which all names hash to 1; so lookups try first 1, then 2, and so on, until the desired name is found.

6.3.6. Hashing Functions and Collisions

It is important that the hashing function generate a uniform distribution of table positions. If the hashing function repeatedly generates the same few table positions, long chains will develop following these positions, and both inserting and searching will be slowed down. Two good hashing functions are discussed below.

Both hashing functions begin by reducing the character string representing the name to a single machine word. If the name contains more characters than fit in a single machine word, all the words containing parts of the name should be added or exclusive or'ed to form one word, which we will call the key.

The first hashing function consists of dividing the key by the table size and using the remainder plus 1 (which will always be between 1 and the table size inclusive) as the table position. If the table size is 2^N, the generated number will simply be the rightmost N bits of the key, with the remaining bits having no effects. The result leads to nonuniform distributions, which is undesirable. The method, known as the **division method**, works best if the table size is a prime number.

The other widely used hashing technique, the **power residue method**, is most appropriate if the table size is 2^N (e.g., if the table occupies exactly one page of a virtual memory). The power residue method consists of regarding the key as a binary integer, multiplying it by a well-chosen constant, and then taking N bits from the middle or low-order end of the product as an integer between 0 and $2^N - 1$. To this integer, 1 is added to give the table position.

Although both hashing functions produce uniform distributions, collisions can still occur, and a method for handling them must be chosen. Collision strategies can be divided into two groups: those that require a new table position to be generated and those that do not. If the first strategy is employed, a way is needed to continue generating new table positions until an empty one is found. To minimize clustering, names having different keys should produce different series of table positions, even if they produce the same initial position.

A good collision method to use with the division hashing function is the **quadratic quotient method** (Bell, 1970). When the key is divided by the table size, both the quotient, Q, and the remainder, R, are saved. The series of table positions is given by

(R + Qi² + i modulo tablesize) + 1

for $i = 0$, 1, 2, and 3. The first four table positions are

(R modulo tablesize) + 1
(R + Q + 1 modulo tablesize) + 1
(R + 4Q + 2 modulo tablesize) + 1
(R + 9Q + 3 modulo tablesize) + 1

It is improbable that two names will generate the same sequence unless they have the same remainder and the same quotient, which is only possible if they have the same key. The quadratic quotient method essentially eliminates clustering.

A good collision method to use with the power residue hashing function is as follows. The product of the key and the constant used in the calculation is used as a new key, and it is multiplied by the constant to generate a new product. The low-order N bits of this product can be converted to a table position. If this position is also full, the new product can be used as a key and the process can be repeated for as many steps as necessary. Note that the same table position may be accidently generated two or more times in the series. The quadratic quotient method can also be employed with a hash table whose size is a power of 2. In this case, R is the initial hash code and Q is 2 (Ackerman, 1974).

An alternative to generating a series of table positions is to link all entries hashing to the same initial value together on a linked list. Figure 6-17 illustrates this kind of chaining schematically. The method is easiest to program if three tables are used: the list header table, the node table, and the name table. The list header table points to the start of each list. The node table entries have three parts—an index into the name table, the value, and the link to the next node on the chain. The name table holds the character strings, as before.

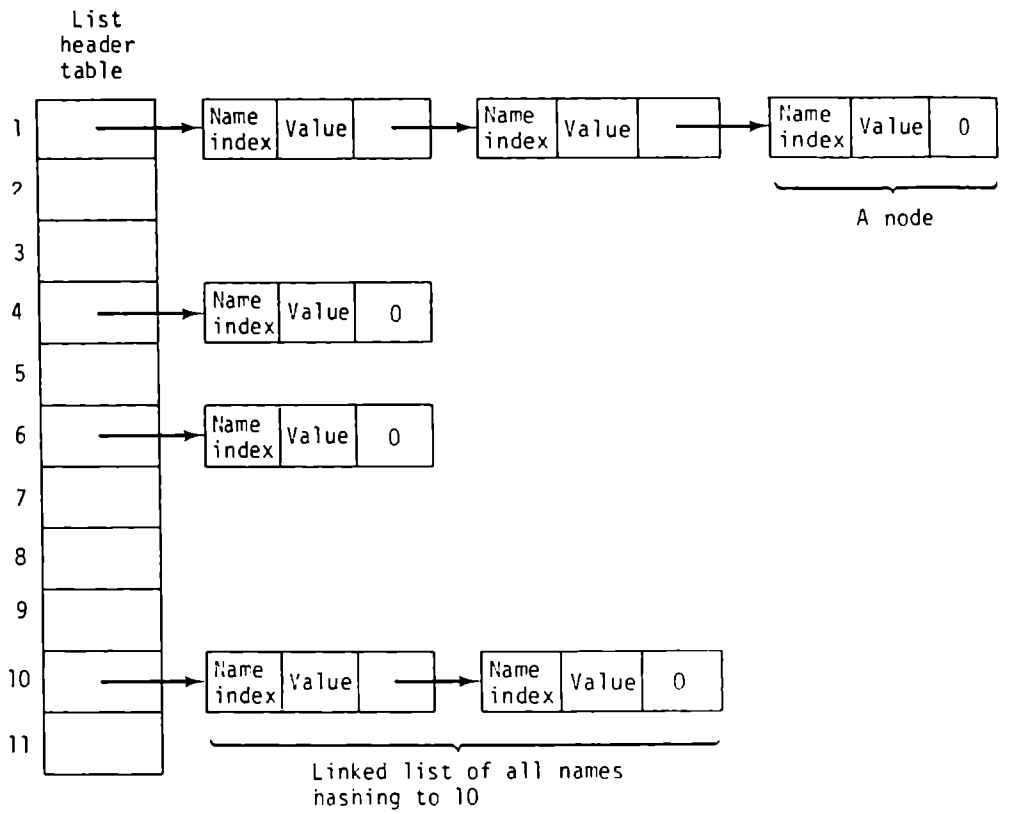


Fig. 6-17. Use of a linked list to chain all entries with the same hash code together.

An approximate, but useful, analysis of the expected performance of hash coding is as follows. If a fraction r of the table is full, the probability of being able to insert a new entry successfully on any attempt is $1 - r$. The probability, $P(k)$, of the insertion requiring exactly k attempts is the probability of $k - 1$ failures times the probability of one success. The expected number of attempts, e, is obtained by summing $kP(k)$ for k from 1 to infinity, yielding

$$e = \frac{1}{1 - r}$$

If twice as much space is allocated to the table as will be needed, r will vary from 0 (at the start of the assembly) to $\frac{1}{2}$ (at the end of the assembly). Early in the assembly there will be few collisions, and the expected number of tries per name to be inserted will be close to 1. When the table becomes half full, the expected number of tries will approach 2. The average number of tries is

$$\frac{1}{r} \ln \frac{1}{1 - r}$$

6.3.7. Comparison of Association Techniques

Three methods for simulating an associative memory have been discussed:

1. An unsorted table with a linear search.
2. A sorted table with a binary search.
3. A hash-coded table.

In all three cases, the aim is to able to associate names (character strings) with values of one or more fields and enter them into a table so that when the name is presented later, the value can be found.

Although only the hashing method was illustrated with variable-length names, all the methods can be used with variable-length names. In all three cases, an extra name table is needed, the main table itself containing a pointer or index into the name table entry instead of the name itself.

The three methods can be compared with respect to several criteria, as shown in Fig. 6-18.

The unsorted table with a linear search is, by any measure of performance, the worst of the three methods overall. It is very easy to program, however, and if speed is unimportant, should be used for that reason. Some assemblers are completely i/o bound. The reading in of statement N + 1 is initiated concurrently with the processing of statement N. If the i/o device is very slow—for example, a paper tape reader reading

Criterion	Linear search on an unsorted table	Binary search on a sorted table	Hash coding
Speed of looking up a name	Slow	Moderate	Fast
Memory utilization	Efficient	Efficient	Wasteful
Ease of programming	Trivial	Moderate	Easy
Ease of removing names	Easy	Fairly easy	Difficult
Is a sort required?	No	Yes	No
Speed of looking up names on pass 1	Slow	Cannot be used before the sort	Fast
Can duplicate names be detected?	No	Yes	Yes

Fig. 6-18. Comparison of three techniques for simulating an associative memory in software.

at 10 characters per second—the processing will always finish long before the next stetement is read in. Thus nothing is to be gained by using an exotic look up technique. The assembly speed is completely determined by the i/o speed.

The choice of a binary search is made to provide faster lookups. Assemblers spend much of their time looking up names. Consequently, unless the assembler is i/o bound, this gain in speed will be noticeable. The principal disadvantage of a binary search is that it requires a sorted table, and sorting takes time. Furthermore, if names need to be looked up on pass one (the EQU statement can equate a symbol to a previously defines symbol), either the symbol table must be resorted before each lookup or a linear search must be used on pass one. Both methods are slow, and there is no avoiding the problem.

Binary search has the advantage that if printing out an alphabetical listing of all the symbols, including the line number on which each was defined and the lines on which each was referenced (which is convenient for the programmer), is required, then the symbol table will have to be sorted in any case, and the sorting time is not really wasted. It should be pointed out, however, that using a hash-coding scheme for entering and looking up names, followed by a sort for printing the listing of symbols, will still be much faster than using a binary search in most cases. If there are 1000 entries, the binary search will require 10 tries to find an entry on the average, whereas hash coding with a half-full table will require less than two tries on the average.

Hash addressing wastes a lot of space compared to the other methods, but it is by far the fastest one. Unless there is some compelling reason to avoid it (e.g., main memory is scarce or the assembler is completely i/o bound), hash coding is clearly the best choice for most assemblers.

6.4. MACROS

Assembly language programmers frequently need to repeat sequences of instructions several times within a program. The most obvious way to do so is simply to write the required instructions, wherever they are needed. If a sequence is long, however, or must be used a large number of times, writing it repeatedly becomes tedious.

An alternative method is to make the sequence into a procedure and call it wherever it is needed. This strategy has the disadvantage of requiring a procedure call instruction and a return instruction to be executed every time a sequence is needed. If the sequences are short—for example, two instructions—but used very frequently, the procedure call overhead may significantly slow the program down. Macros provide an easy and efficient solution to the problem of repeatedly needing the same or nearly the same sequences of instructions.

6.4.1. Macro Definition, Call, and Expansion

A **macro definition** is a method for giving a name to a piece of program. After a macro has been defined, the programmer can write the macro name instead of the piece of program. A macro is, in effect, an abbreviation for a piece of text. Figure 6-19 shows assembly language programs for the 370, Cyber 70, and PDP-11, which ex-

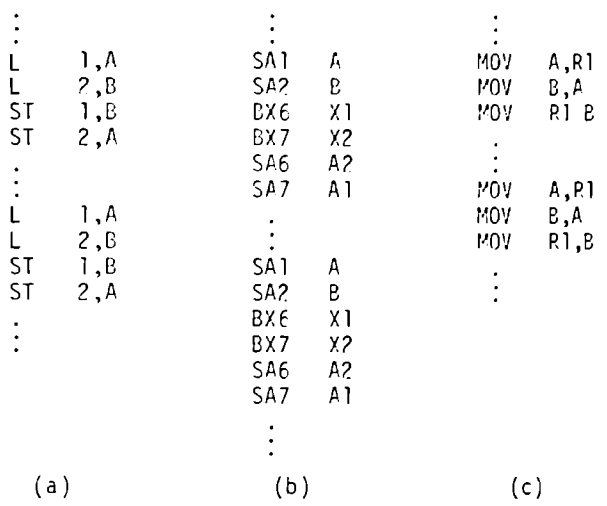


Fig. 6-19. Assembly language statements for interchanging the variables A and B. Each program contains the same sequence twice. (a) IBM 370. (b) Cyber 70. (c) PDP-11.

change the contents of the variables A and B. These sequences could be defined as macros, as shown in Fig. 6-20. After the definition of Fig. 6-20(a), every occurrence of SWAP in the 370 assembly language program causes it to be replaced by the four lines

```
L     1,A
L     2,A
ST    1,B
ST    2,A
```

The programmer has defined SWAP as an abbreviation for the four statements shown above.

Although the three assemblers have slightly different notation for defining macros, all three require the same basic parts in a macro definition:

1. A macro header giving the name of the macro being defined.
2. The text comprising the body of the macro.
3. A pseudoinstruction marking the end of the definition.

When the assembler encounters a macro definition, it saves it in a macro definition table for subsequent use. From that point on, whenever the name of the macro (SWAP in the example of Fig. 6-20) appears as an opcode, the assembler replaces it by the macro body. The use of a macro name as an opcode is known as a **macro call**, and its replacement by the macro body is called **macro expansion**.

Macro expansion occurs during the assembly process and not during execution of the program. This point is very important. The programs of Fig. 6-19 without macros and of Fig. 6-20 with macros will produce precisely the same machine language code.

```
MACRO
SWAP
L     1,A
L     2,B
ST    1,B
ST    2,A
MEND

.
.
.
SWAP
.
.
.
SWAP
.
.
.
```

(a)

```
SWAP  MACRO
      SA1     A
      SA2     B
      BX6     X1
      BX7     X2
      SA6     A2
      SA7     A1
      ENDM
      .
      .
      .
      SWAP
      .
      .
      .
      SWAP
      .
      .
      .
```

(b)

```
.MACRO   SWAP
MOV      A,R1
MOV      B,A
MOV      R1,B
.ENDM
.
.
.
SWAP
.
.
.
SWAP
.
.
.
```

(c)

Fig. 6-20. Macro definitions for a macro SWAP that interchanges A and B, and their use. (a) IBM 370. (b) Cyber 70. (c) PDP-11.

Looking only at the machine language program, it is impossible to tell whether or not any macros were involved in its generation, because the macro expansion has been completed and the macro definitions discarded by the time the program has been completely assembled.

Macro calls should not be confused with procedure calls. The basic difference is that a macro call is a signal to the assembler to replace the macro name with the macro

Item	Macro call	Procedure call
When is the call made?	During assembly	During execution of the object program
Is the body inserted into the object program every place the name appeared?	Yes	No
Is a procedure call instruction inserted into the object program and later executed?	No	Yes
Must a return instruction be used to return control to the statement following the call?	No	Yes
How many copies of the body appear in the object program?	One for each macro call	1

Fig. 6-21. Comparison of macro calls and procedure calls.

body. A procedure call is a machine instruction that is inserted into the object program and that will later be executed to call the procedure. Figure 6-21 compares macro calls with procedure calls.

Although macros are generally expanded on pass one of the assembly process, it may be conceptually simpler to think of the assembler as having an extra pass before pass one during which macro definitions are saved and macros are expanded. In this view, the source program is read in and is then transformed into another program from which all macro definitions have been removed, and in which all macro calls have been replaced by their bodies. The resulting output, an assembly language program containing no macros at all, is then fed into the assembler.

It is important to keep in mind that a program is a string of characters, including letters, digits, spaces, punctuation marks, and "carriage returns" (change to a new card). Macro expansion consists of replacing certain substrings of this string with other character strings. A macro facility is a method of manipulating character strings without regard to their meaning.

6.4.2. Macros with Parameters

The macro facility just described can be used to shorten programs in which precisely the same sequence of instructions occurs repeatedly. Frequently, however, a program contains several sequences of instructions that are almost, but not quite, identical, as in Fig. 6-22. Here the first sequence exchanges A and B, and the second sequence exchanges C and D.

Macro assemblers handle the case of nearly identical sequences by allowing macro definitions to provide formal parameters and by allowing macro calls to supply actual parameters. When a macro is expanded, each formal parameter appearing in the

```
.                .                .
.                .                .
.                .                .
L    1,A         SA1  A           MOV  A,R1
L    2,B         SA2  B           MOV  B,A
ST   1,B         BX6  X1          MOV  R1,B
ST   2,A         BX7  X2            .
                 SA6  A2            .
.                SA7  A1            .
.                                 MOV  C,R1
.                .                MOV  D,C
L    1,C         .                MOV  R1,D
L    2,D         .                  .
ST   1,D         SA1  C             .
ST   2,C         SA2  D             .
.                BX6  X1
.                BX7  X2
                 SA6  A2
                 SA7  A1
                 .
                 .
(a)              (b)              (c)
```

Fig. 6-22. Nearly identical sequences of statements. (a) IBM 370. (b) Cyber 70. (c) PDP-11.

```
.
.
.
MACRO
CHANGE   &P,&Q
L        1,&P
L        2,&Q
ST       1,&Q
ST       2,&P
MEND
   .
   .
   .
CHANGE   A,B
   .
   .
   .
CHANGE   C,D
   .
   .
   .
```

(a)

```
CHANGE   MACRO    P,Q
         SA1      P
         SA2      Q
         BX6      X1
         BX7      X2
         SA6      A2
         SA7      A1
         ENDM
            .
            .
            .
         CHANGE   A,B
            .
            .
            .
         CHANGE   C,D
            .
            .
            .
```

(b)

```
.MACRO   CHANGE,P,Q
MOV      P,R1
MOV      Q,P
MOV      R1,Q
.ENDM
   .
   .
CHANGE   A,B
   .
   .
CHANGE   C,D
   .
   .
```

(c)

Fig. 6-23. Macros with parameters. The machine language programs generated are identical to those of Fig. 6-22. (a) IBM 370. (b) Cyber 70. (c) PDP-11.

macro body is replaced by the corresponding actual parameters. The actual parameters are placed in the operands field of the macro call. Figure 6-23 shows the programs of Fig. 6-22 rewritten using macros with parameters. The symbols P and Q are the formal parameters. Each occurrence of P within a macro body is replaced by the first actual parameter when the macro is expanded. Similarly, Q is replaced by the second actual parameter. In the macro call

```
CHANGE A,B
```

A is the first actual parameter and B is the second actual parameter. The 370 assembler requires an ampersand to indicate the start of the formal parameters. The other two assemblers do not require any special marker. Other than that, all three macro facilities have similar syntaxes. For that reason, we will simply use the PDP-11 as an example from now on.

6.4.3 Conditional Macro Expansion

Most macro assemblers provide a method for expanding macro calls in different ways, depending on the actual parameters. This feature, called **conditional assembly**, or **conditional macro expansion**, can be used to optimize the code generated by the macros. For example, if the CHANGE macro is called with two identical actual parameters, such as

```
CHANGE A,A
```

there is no need to produce any output at all. A PDP-11 macro that produces an expansion only if the actual parameters are different is

```
.MACRO EXCHNG,P,Q
.IF DIF,<P>,<Q>
MOV P,R1
MOV Q,P
MOV R1,Q
.ENDC
.ENDM
```

The pseudoinstruction .IF takes as parameters a specification of the condition under which the following statements are to be expanded. The condition DIF followed by two parameters enclosed in angular brackets means that the statements between the .IF and the .ENDC (END Conditional) are to be output if and only if the two character strings are different. The relation IDN tests to see if the two character strings are identical, rather than different. Relations for testing numerical values exist as well. If the first parameter is EQ, NE, LT, GT, LE, or GE, the second parameter's numerical value is compared to zero to see if it is equal, not equal, less than, greater than, less than or equal, or greater than or equal respectively.

The macro call

```
EXCHNG X,Y
```

is expanded in two steps. First, the macro itself is expanded, with P replaced by X and Q replaced by Y. This step yields

```
.IF DIF,<X>,<Y>
MOV X,R1
MOV Y,X
MOV R1,Y
.ENDC
```

Then the .IF is processed. Because <X> and <Y> are different characters, the text between .IF and .ENDC is output. The final result of expanding EXCHNG X,Y is

```
MOV X,R1
MOV Y,X
MOV R1,Y
```

On the other hand,

```
EXCHNG X,X
```

produces no expansion at all. The macro call EXCHNG X,X is removed from the program, and nothing is put in its place, which is precisely what is desired.

Any statements may be written between .IF and .ENDC, including another .IF - .ENDC pair. When conditionals are nested, the expansion begins at the outermost .IF and works its way inward. Whenever a .IF whose condition is not true is encountered (e.g., DIF with identical strings or IDN with different strings), all the statements up to the matching .ENDC are skipped.

As an example of nested conditionals, we will write a macro JUGGLE to exchange two variables, even if one of the variables is R1. You should note that the macro call

```
EXCHNG COUNT,R1
```

produces

```
MOV COUNT,R1
MOV R1,COUNT
MOV R1,R1
```

which does not exchange COUNT and R1. The correct macro, assuming that no call uses both R1 and R2, is

```
.MACRO JUGGLE,P,Q
.IF DIF,<P>,<Q>
    IF IDN,<P>,<R1>
        MOV P,R2
        MOV Q,P
        MOV R2,Q
    .ENDC; THIS MATCHES .IF IDN,<P>,<R1>
    .IF IDN,<Q>,<R1>
        MOV P,R2
        MOV Q,P
        MOV R2,Q
    .ENDC ; THIS MATCHES .IF IDN,<Q>,<R1>
    .IF DIF,<P>,<R1>
        .IF DIF<,Q>,<R1>
            MOV P,R1  ; NOTE THAT THESE 3 STATEMENTS ARE ONLY
            MOV Q,P   ; EXPANDED IF P IS DIFFERENT FROM R1 AND
            MOV R1,Q  ; Q IS DIFFERENT FROM R1
        .ENDC ; THIS MATCHES .IF DIF,<Q>,<R1>
    .ENDC ; THIS MATCHES .IF DIF,<P>,<R1>
.ENDC ; THIS MATCHES .IF DIF,<P>,<Q>
.ENDM ; THIS ENDS THE MACRO DEFINITION
```

6.4.4 Nested Macro Calls

Since a macro call is used in the same way as an ordinary machine instruction, it is natural to expect a macro call to be allowed within a macro body. Most macro assemblers do, in fact, permit this nesting. As an illustration of a macro called from within another macro, consider a macro CIRCLE that performs a left-circular permutation of its parameters (not including register R2). For example, if the variables A, B, and C have the values 1, 2, and 3, respectively, before the left-circular permutation, they will have the values 2, 3, and 1, respectively, after it.

```
.MACRO CIRCLE,A,B,C
JUGGLE A,B
JUGGLE B,C
.ENDM
```

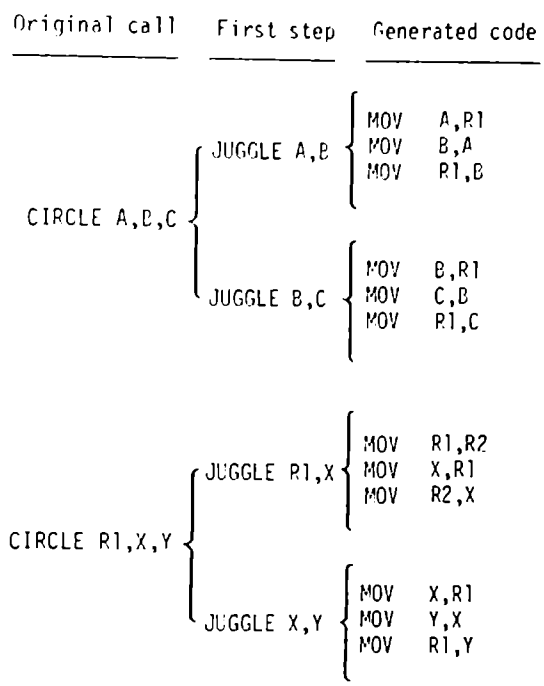


Fig. 6-24. Two examples of the expansion of a macro whose body contains other macro calls.

Figure 6-24 shows two expansions of circle. This definition of CIRCLE is not optimum. Writing an optimizing macro for circular permutation is left to the reader.

The preceding example demonstrates a two-step macro expansion. Macros with 3, 4, or more steps in their expansion are also allowed. Simple macros can be used in the bodies of more complicated macros, which, in turn, can be used to construct still more complicated macros. In this way, a programmer can, in effect, define a new level, all of whose instructions are macros. Macro expansion is another translation technique for implementing multilevel machines.

6.4.5. Recursive Macro Calls

An interesting situation arises when a macro call appearing within a macro definition calls the macro being defined. Such a macro is known as a **recursive macro**, in analogy with a recursive procedure, which also calls itself. When the definition of a macro, recursive or otherwise, is encountered by the assembler, the body of the macro is copied character by character into the assembler's macro definition table without even being examined to see if it contains assembly language statements, let alone a recursive macro call. When a macro call is encountered, the call is replaced by the stored definition, with the formal parameters replaced by the actual parameters. The assembler processes the macro body just as though it were part of the original source program.

Bearing in mind that macro definition and expansion consist of character manipulation and nothing more, we can see how a recursive macro can and cannot be used. When used properly, recursive macros are a powerful programming tool; when

misused, recursive macros can put an assembler into an infinite loop. Consider the following macro:

```
.MACRO REPEAT,N
REPEAT N
.WORD N
.ENDM
```

The macro call REPEAT 4 is expanded by replacing the call with the body. This expansion yields two statements

```
REPEAT 4
.WORD 4
```

The first is another macro call, and the second is a pseudoinstruction that assembles one machine word containing the constant 4. The next step consists of expanding REPEAT 4 again. This process continues indefinitely, since each expansion produces exactly one more macro call to be expanded. As a result, the assembler gets stuck in a never-ending loop, as shown in Fig. 6-25.

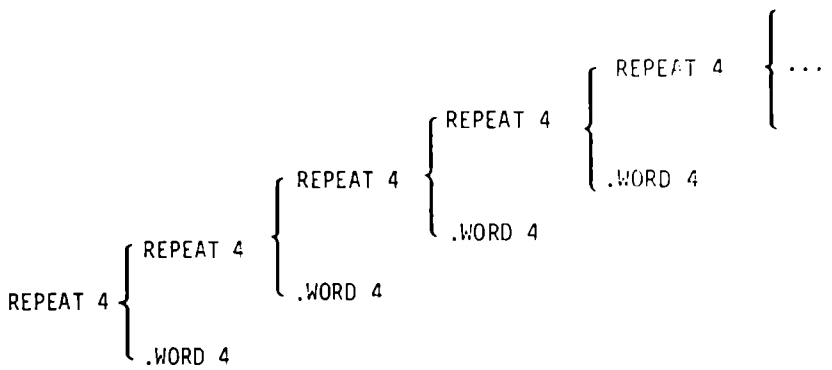


Fig. 6-25. A call of the REPEAT macro will put the assembler into an infinite loop.

The conditional assembly pseudoinstruction .IF can be used to define recursive macros whose expansion is finite and potentially useful. For instance, consider the macro TABLE

```
.MACRO TABLE,COUNT
.IF NE,COUNT
TABLE COUNT-1
.ENDC
.WORD COUNT
.ENDM
```

Figure 6-26 shows the expansion of TABLE, which generates a table of values, one per word, from 0 to the value of the actual parameter. The first step in expanding the macro call in Fig. 6-26(a) consists of replacing the call by the body, substituting the actual parameter for the formal parameter, as shown. The four generated statements are then processed, just as though they had been part of the original program.

The .IF pseudoinstruction tests the expression 4 to see if its numerical value is

```
(a)   TABLE 4      .IF NE, 4
                   TABLE 4-1
                   .ENDC
                   .WORD 4

                   .IF NE, 4-1
      TABLE 4-1    TABLE 4-1-1
(b)   .WORD 4      .ENDC
                   .WORD 4-1

                   .IF NE, 4-1-1
      TABLE 4-1-1  TABLE 4-1-1-1
(c)   .WORD 4-1    .ENDC
      .WORD 4      .WORD 4-1-1-1

                   .IF NE, 4-1-1-1
      TABLE 4-1-1-1TABLE 4-1-1-1-1
(d)   .WORD 4-1-1  .ENDC
      .WORD 4-1    .WORD 4-1-1-1
      .WORD 4

                   .IF NE, 4-1-1-1-1
      TABLE 4-1-1-1-1TABLE 4-1-1-1-1
      .WORD 4-1-1-1.ENDC
(e)   .WORD 4-1-1  .WORD 4-1-1-1-1
      .WORD 4-1
      .WORD 4

      .WORD 4-1-1-1-1-1 ----> 000000
      .WORD 4-1-1-1-1   ----> 000001
(f)   .WORD 4-1-1-1     ----> 000002
      .WORD 4-1         ----> 000003
      .WORD 4           ----> 000004
```

Fig. 6-26. (a)–(e) Expansion of TABLE 4. (f) The resulting machine code.

equal to 0. Since the relation NE means "not equal to zero," the following statement will only be expanded if the expression is nonzero, which it is. The result of this expansion is shown at the left in Fig. 6-26(b)—namely, two statements, a macro call of table with 4–1 as actual parameter, and a .WORD pseudoinstruction. Note that the actual parameter in the call of TABLE is the three-character string "4–1", not the number "3".

TABLE 4–1 is expanded by replacing the call by the body, with the characters "4–1" being substituted for COUNT. This process is repeated until .IF is given the expression "4–1–1–1–1" to evaluate. Since this expression is numerically 0, the test fails and the statement following is not expanded. The result of TABLE 4 is five machine words, containing 0, 1, 2, 3, and 4.

6.4.6. Nested Macro Definitions

A macro name is simply an abbreviation for some text. Consequently, a macro body may contain any statements that are valid outside a macro, including definitions of other macros. Nested macro definitions cause one minor problem—namely, when a macro definition is encountered. The macro assembler stores subsequent text until

it encounters an .ENDM statement. If macro definitions are nested, the .ENDM of the inner macro definition will inadvertently terminate the definition. To avoid confusion, most macro assemblers require that the .ENDM of an inner definition contain the name of the macro definition being terminated.

A macro definition occurring inside another macro definition does not take effect until the macro in which it has been defined has been called and expanded. This is a consequence of the way macros work. When a macro definition is found, its body is stored without being examined. As a result, definitions occurring inside are not seen. When the surrounding macro is called, its body is scanned by the assembler, which causes the embedded definitions to be found and activated.

6.4.7. Implementation of a Macro Facility in an Assembler

To implement a macro facility, an assembler must be able to perform two functions: save macro definitions and expand macro calls. We will examine these functions in turn.

The assembler must maintain a table of all macro names and, along with each name, a pointer to its stored definition so that it can be retrieved when needed. Some assemblers have a separate table for macro names, and some have a combined opcode table in which all machine instructions, pseudoinstructions, and macro names are kept.

When a macro definition is encountered, an entry is made in the macro or combined opcode table giving the name of the macro, the number of formal parameters, and a pointer to another table—the macro definition table—where the macro body will be kept. A list of the formal parameters is also constructed at this time for use in processing the definition. The macro body is then read and stored in the macro definition table. To save space, trailing blanks are not stored. Formal parameters occurring within the body are not stored but instead are replaced by a special marker (to indicate formal parameter) followed by the parameter number, in order to simplify the macro expansion process. As an example, the internal representation of the macro definition of Fig. 6-23(c), with a tilde as parameter marker is shown below.

```
MOV ~1,R1↵MOV ~2, ~1↵MOV R1, ~2↵
```

Here ↵ represents a new line. Within the macro definition table, the macro body is simply a character string.

During pass one of the assembly, opcodes are looked up and macros expanded. Assemblers normally have a procedure that is called to read the next line of the program. Usually this procedure gets the next line from the input file. However, when a macro is being expanded, it takes its input from the macro definition table. To keep track of where it must fetch the next line from, the READ procedure maintains a stack, the top entry of which tells whether a macro is being expanded, and, if so, which one, and what line is next. Figure 6-27 illustrates the expansion of

```
CIRCLE MOUSE,CAT,DOG
```

using the definition of CIRCLE from Section 6.4.4.

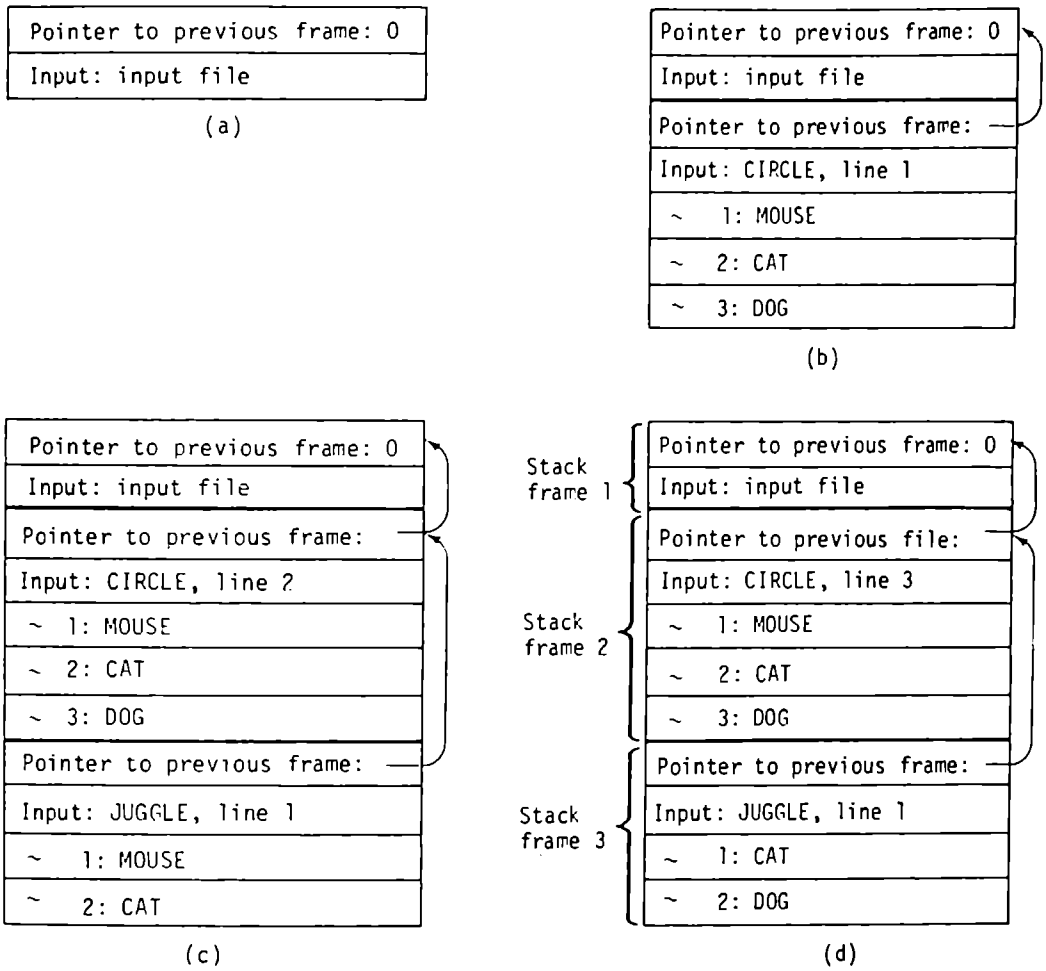


Fig. 6-27. The expansion stack at several points during the expansion of the macro call CIRCLE MOUSE, CAT, DOG.

Each stack frame contains

1. The stack pointer of the previous frame.
2. Information telling where the next input line is to come from.
3. The actual parameters used in this expansion (or pointers to them).

When assembly starts, the only stack frame specifies that input is coming from the input file, as shown in Fig. 6-27(a). When the assembler recognizes the first call of CIRCLE, it creates a new frame on the stack, as shown in Fig. 6-27(b). After the call of JUGGLE is recognized, another frame is added to the stack, as shown in Fig. 6-27(c). Once the expansion of JUGGLE is finished, the top frame is removed from

the stack. Just prior to the expansion of the second call of JUGGLE, the stack is as shown in Fig. 6-27(d).

Wherever the READ procedure encounters the parameter marker, it substitutes the actual parameter in place of the parameter marker. It should now be clear why the formal parameters were replaced by a special marker and a parameter number (and why the 370 formal parameters must begin with an ampersand): to simplify recognition of formal parameters during macro expansion.

6.5. LINKING AND LOADING

Most programs consist of more than one procedure. Compilers and assemblers generally translate one procedure at a time and put the translated output in secondary memory. Before the program can be run, all the translated procedures must be found and linked together properly. If virtual memory is not available, the linked program must be explicitly loaded into main memory as well. Programs that perform these functions are called by various names, including **linker**, **loader**, **linking loader**, and **linkage editor**. The complete translation of a source program requires two steps, as shown in Fig. 6-28:

1. Compilation or assembly of the source procedures.
2. Linking of the object modules.

The translation from source procedure to object module represents a change of level because the source language and target language have different instructions and notation. The linking process, however, does not really represent a change of level, for both the linker's input and the linker's output are programs for the same machine. The linker's function is to collect procedures translated separately and link them together properly to be run as a unit, usually called an **absolute load module**. The

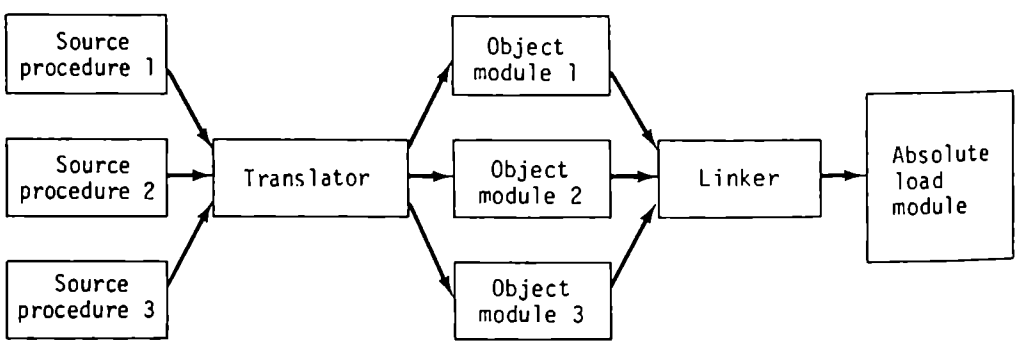


Fig. 6-28. Generation of an absolute load module from a collection of independently translated source procedures requires using a linker.

loader's function is to load the absolute load module into main memory. These functions are often combined.

There is a good reason why compilers and assemblers translate each source procedure as a separate entity. If a compiler or assembler read a series of source procedures and directly produced a ready-to-run machine language program, then changing one statement in one source procedure would require that all the source procedures be retranslated.

If the separate-object-module method of Fig. 6-28 is used, it is only necessary to retranslate the modified procedure and not the unchanged ones, although it is necessary to relink all the object modules again. Linking is usually much faster than translating, however; thus the two-step process of translating and linking can save a great deal of time during the development of a large program. Short student jobs do not need the flexibility of separate translation. As a result, compilers have been written for certain languages (e.g., WATFOR and PL/C) that produce ready-to-run machine language programs.

6.5.1. Tasks Performed by the Linker

At the start of pass one of the assembly process, the instruction location counter is set to 0. This step is equivalent to assuming that the object module will be located at (virtual) address 0 during execution. Figure 6-29 shows four object modules for the Cyber 70. Each module begins with a JUMP instruction to an SX1 instruction within the module. In order to run the program, the loader brings the object modules into main memory, as shown in Fig. 6-30(a). On the Cyber 70, the bottom part of memory is used for communication with the operating system, so the first object module cannot be loaded at 0.

The program of Fig. 6-30(a), although loaded into main memory, is not yet ready for execution. Consider what would happen if execution began with the instruction at the beginning of module A. The program would not jump to the SX1 1 instruction as it should, because that instruction is now at 300. In fact, all memory reference instructions will fail to operate properly for the same reason.

This problem occurs because each object module in Fig. 6-29 represents a separate address space. On a machine with a segmented address space, such as provided by MULTICS, each object module can have its own address space by placing it in its own segment. On a machine with a linear, one-dimensional memory, such as the Cyber 70, the object modules must be merged into a single address space. The two-dimensional nature of the MULTICS virtual memory eliminates the need for merging object modules and greatly simplifies the task of the linker. The separate address spaces of the object modules must also be merged on a machine with a paged, one-dimensional virtual memory.

Furthermore, the procedure call instructions in Fig. 6-30(a) will not work either. At address 400, the programmer had intended to call object module B; but since each procedure is translated by itself, the assembler has no way of knowing what address

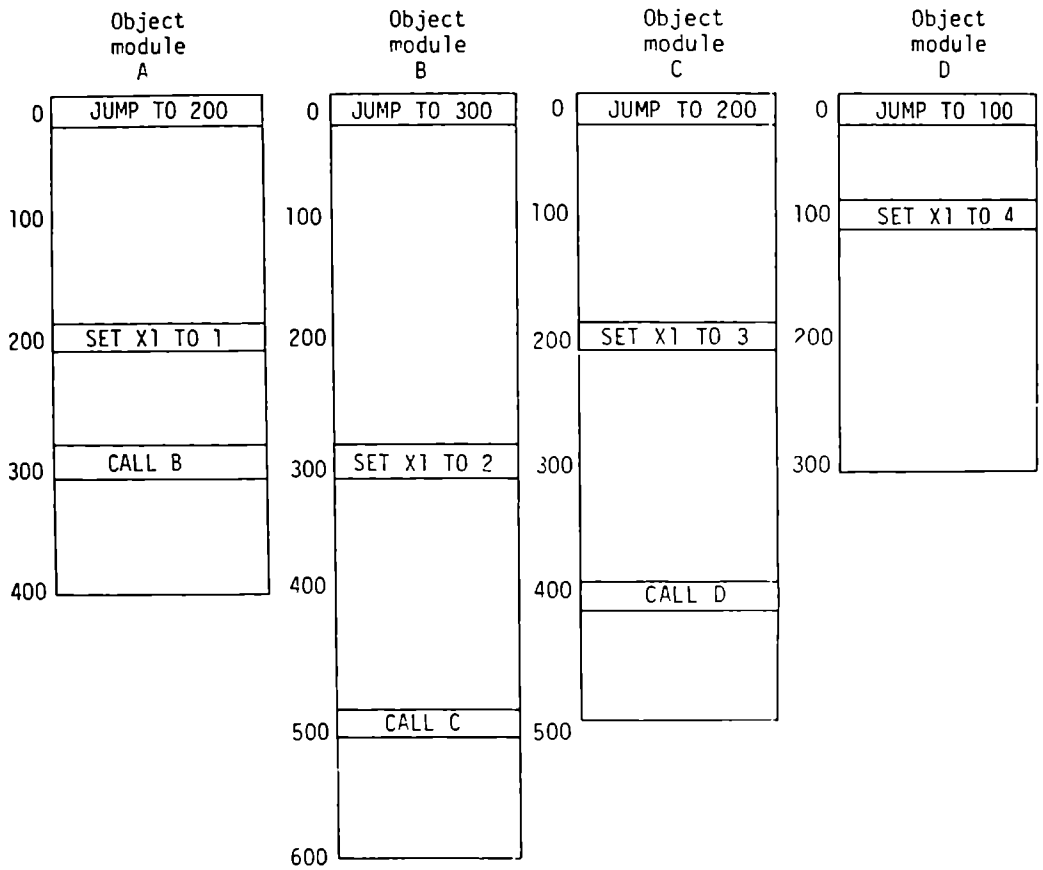


Fig. 6-29. Each object module has its own address space, starting at 0.

to insert into the CALL instruction. The address of object module B is not known until linking time. Both of these problems can be solved by the linker.

The linker merges the separate address spaces of the object modules into a single linear address in the following steps:

1. It constructs a table of all the object modules and their lengths.
2. Based on this table, it assigns a load address to each object module.
3. It finds all the instructions that contain a memory address, and to each adds a **relocation constant** equal to the starting address of the module in which it is contained.
4. It finds all the instructions that reference other procedures and inserts the address of these procedures in place.

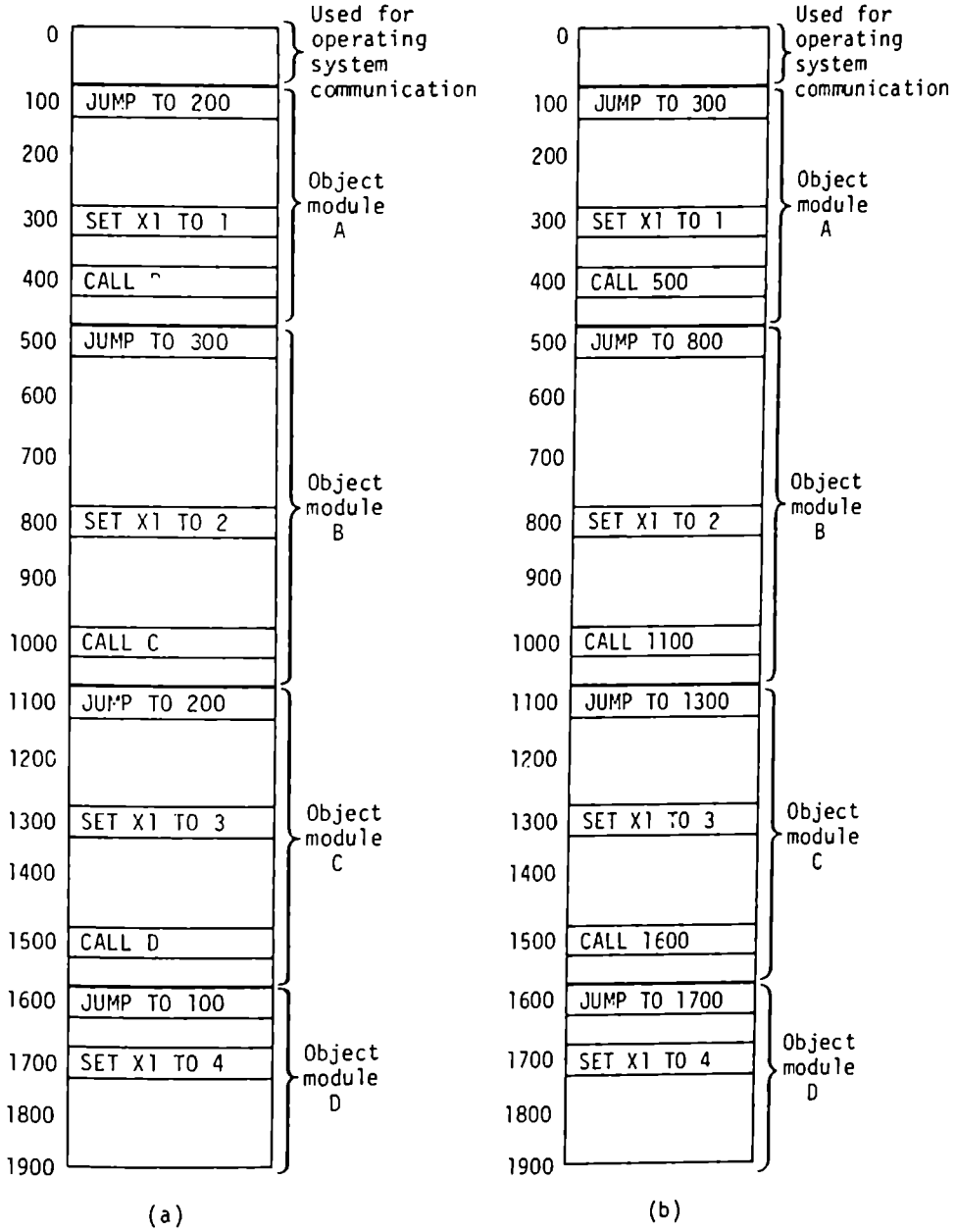


Fig. 6-30. (a) The object modules of Fig. 6-29 after being loaded but before linked (b) The same object modules after linking is complete. This is an absolute load module, and is ready to run.

The object module table constructed in step 1 is shown for the modules of Fig. 6-30:

Module	Length	Starting Address
A	400	100
B	600	500
C	400	1100
D	300	1600

Figure 6-30(b) shows how the address space of Fig. 6-30(a) looks after the linker has performed these steps.

Base register addressing speeds up linking by reducing the number of instructions that must be relocated. Since the calling sequence loads a base register with the origin of the procedure being called, that procedure can reference its instructions and data by specifying the displacement from the start of the procedure, which is independent of where the procedure is loaded. The PDP-11's self-relative addressing also eliminates the need for the linker to add relocation constants to instructions.

6.5.2. Structure of an Object Module

Object modules contain six parts, as shown in Fig. 6-31. The first part contains the name of the module, certain information needed by the linker, such as the lengths of the various parts of the object module, and sometimes the date of assembly.

The second part of the object module is a list of the symbols defined in the module that other modules may reference, along with their values. For example, if the module

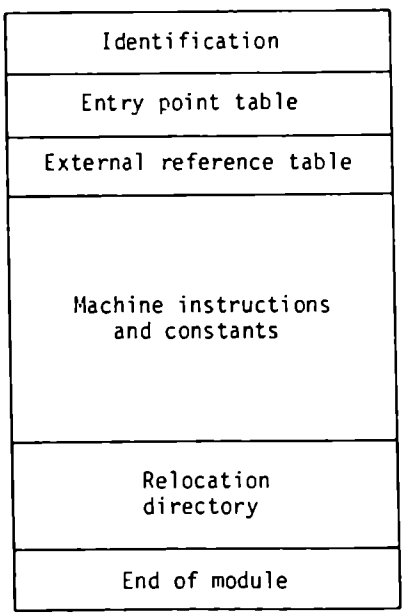


Fig. 6-31. The internal structure of an object module produced by a translator.

consists of a procedure named BIGBUG, then the entry point table will contain the character string "BIGBUG" followed by the address to which it corresponds. The assembly language programmer indicates which symbols are to be declared as entry points by using a pseudoinstruction (see Fig. 6-2).

The third part of the object module consists of a list of the symbols used in the module but defined in other modules, together with a list of which machine instructions use which symbols. The linker needs the latter list in order to be able to insert the correct addresses into the instructions that use external symbols. A procedure can call other independently translated procedures by declaring the names of the called procedures to be external. The assembly language programmer indicates which symbols are to be declared as external by using a pseudoinstruction (see Fig. 6-2). On some computers, the entry point table and the external reference table are combined into a single table.

The fourth part of the object module is the assembled code and constants. This is the only part of the object module that will be loaded into memory to be executed. The other five parts will be used by the linker and then discarded before execution begins.

The fifth part of the object module is the relocation dictionary. As shown in Fig. 6-30, instructions that contain addresses must have a relocation constant added. Since the linker has no way of telling by inspection which of the data words in part 4 contain machine instructions and which contain constants, information about which addresses are to be relocated is provided in this table. The information may take the form of a bit table, with one bit per potentially relocatable address, or an explicit list of addresses to be relocated.

The sixth part is an end-of-module indication, a checksum to catch errors made while reading the module, and sometimes the address at which to begin execution.

Most linkers require two passes. On pass one the linker reads all the object modules and builds up a table of module names and lengths, plus a global symbol table consisting of all entry points and external references. On pass two the object modules are read, relocated, and linked one module at a time. Some linkers assemble the absolute load module directly in main memory so that it can be executed, whereas others write it into secondary memory along with the tables so that it can be merged with other object modules later.

6.5.3. Binding Time and Dynamic Relocation

In a time-sharing system, a program can be read into main memory, run for a little while, written back into secondary memory, and then read back into main memory to be run again. In a large system with many programs, it will be very difficult to ensure that a program is read back into the same locations every time.

Figure 6-32 shows what would happen if the already relocated program of Fig. 6-30(b) were reloaded at address 400 instead of address 100, where the linker put it originally. All the memory addresses are incorrect; moreover, the relocation information has long since been discarded. Even if the relocation information were still

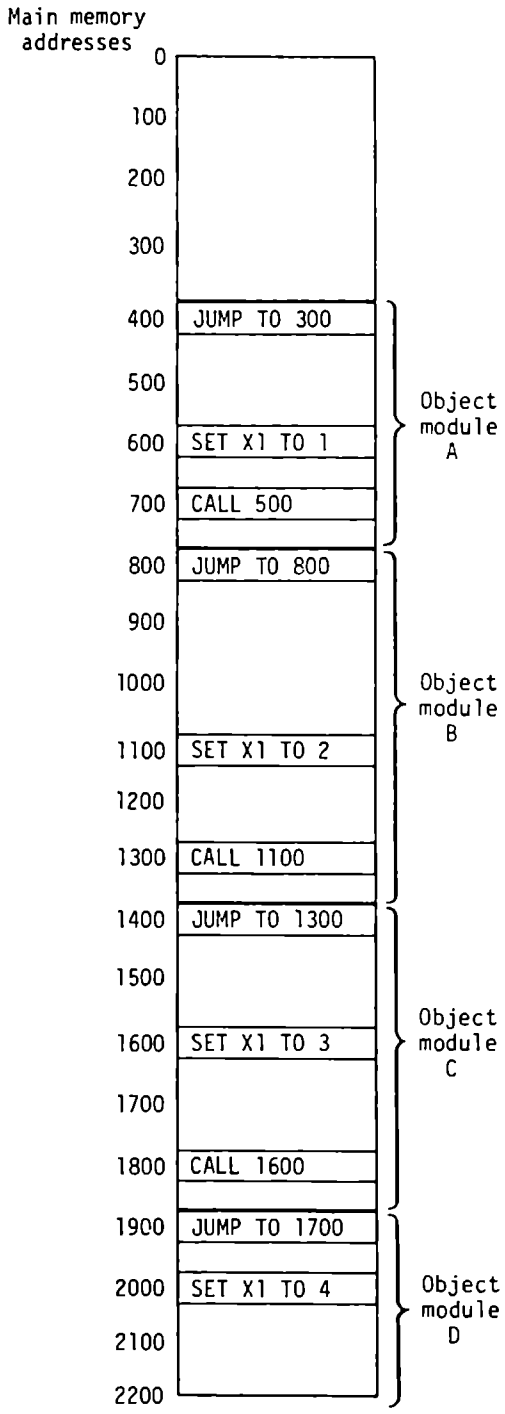


Fig. 6-32. The absolute load module of Fig. 6-30(b) moved up 300 addresses.

available, the cost of having to relocate all the addresses every time the program was swapped would be orders of magnitude too high.

The problem of moving programs that have already been linked and relocated is intimately related to the time at which the final binding of symbolic names onto absolute physical memory addresses is completed. When a program is written, it contains symbolic names for memory addresses. For example, a JUMP to L in the Cyber 70 assembly language is written

```
JP L
```

The time at which the actual main memory address corresponding to L is determined is called the **binding time**. There are at least six possibilities for the binding time:

1. When the program is written.
2. When the program is translated.
3. When the program is linked, but before it is loaded.
4. When the program is loaded.
5. When a base register used for addressing is loaded.
6. When the instruction containing the address is executed.

If an instruction containing a memory address is moved subsequent to binding, it will be incorrect. If the translator produces an absolute load module as output, the binding has occurred at translation time, and the program must be run at the address the translator expected it to be run at. The linking method described in the previous section binds symbolic names to absolute addresses during linking, which is why moving it to a different address subsequent to linking fails, as shown in Fig. 6-32.

Two related issues are involved here. First, there is the question of when symbolic names are bound to virtual addresses. Second, there is a question of when virtual addresses are bound to physical addresses. Only when both operations have taken place is binding complete. When the linker merges the separate address spaces of the object modules into a single linear address space, it is, in fact, creating a virtual address space. The relocation and linking serve to bind symbolic names onto specific virtual addresses. This point is true whether or not virtual memory is being used.

Assume, for the moment, that the address space of Fig. 6-30(b) were paged, although actually the Cyber 70 does not have virtual memory. It is clear that the virtual addresses corresponding to the symbolic names A, B, C, and D have already been determined, even though their physical main memory addresses will depend on the contents of the page table at the time they are used. An absolute load module is really a binding of symbolic names onto virtual addresses.

Any mechanism that allows the mapping of virtual addresses onto physical main memory addresses to be changed easily will facilitate moving programs around in main memory, even after they have been bound to a virtual address space. One such mechanism is paging. After a program has been moved in main memory, only its page table need be changed, not the program itself.

A second mechanism is the use of a relocation register, as is the case on the Cyber 70. The Cyber 70 hardware contains a single relocation register shared by all programs. When a program is to be executed, the operating system sets the relocation register to the physical main memory address of the start of the program. All memory addresses have the relocation register added to them before being sent to the memory. The entire relocation process is transparent to the user programs. They do not even know it is occurring and have no way of interfering with it. When a program is moved, the operating system must update the relocation register. This mechanism is less general than paging, since the entire program must be moved as a unit.

A third mechanism is possible on machines that can use the program counter as a base register, such as the PDP-11. Whenever a program is moved in main memory, only the program counter need be updated. Furthermore, if virtual memory is being used, a program can be "slid up" from one virtual address to another and still work correctly, provided that the program counter is updated. In contrast, a program on a machine that uses paging or a relocation register but that does not use the program counter as a base register can be moved in physical memory, but it cannot be moved to a different virtual address after linking. The ability to load a procedure anywhere within the virtual address space without having to modify it is particularly useful for library procedures.

At first, you might think that base register addressing also solves the dynamic relocation problem, but, in fact, it does not. The procedure call sequence on a base register machine loads a base register with the address of the procedure called. This register is then used as a base register for addresses within the procedure. As long as the procedure is not moved subsequent to being called, everything works fine. However, if the procedure is moved during its execution (a normal situation in a time-sharing system), the base register will no longer contain the correct origin.

Furthermore, it is not possible for the operating system to move a program and update its base register because a program may use several base registers. The operating system has no way of knowing which of the general registers are being used as base registers and which are being used for holding data. This problem is solved on the 370 by using virtual memory. However, on the earlier 360 machines, which did not have virtual memory (except the 360/67), there was no solution and programs could not be moved after being loaded.

6.5.4. Dynamic Linking

The linking method discussed in Section 6.5.1 has the property that all procedures that a program might call are linked before the program can begin execution. On a computer with virtual memory, completing all linking before beginning execution does not take advantage of the full capabilities of the virtual memory. Many programs have procedures that are only called under unusual circumstances. For example, compilers have procedures for compiling rarely used statements, plus procedures for handling error conditions that seldom occur.

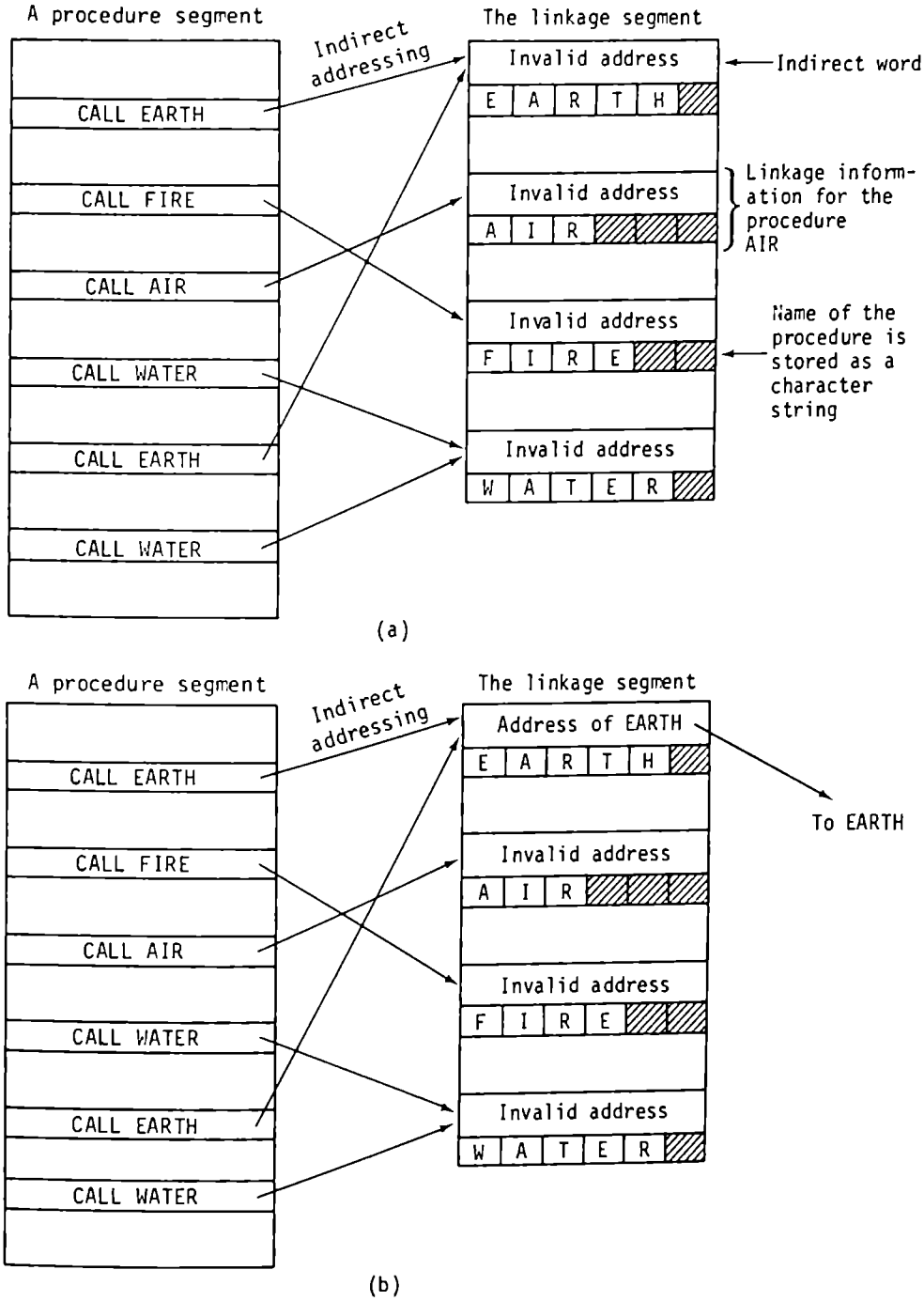


Fig. 6-33. Dynamic linking. (a) Before EARTH is called. (b) After EARTH has been called and linked.

A more flexible method for linking separately compiled procedures is to link each procedure at the time it is first called. This process is known as **dynamic linking**, and it is used by MULTICS. The location of each compiled procedure in secondary memory is recorded somewhere (e.g., the owner's file directory) so that the linker can find it when needed. Associated with each program is a segment, called the **linkage segment**, which contains one block of information for each procedure that might be called. This block of information starts with a word reserved for the virtual address of the procedure, and it is followed by the procedure name stored as a character string.

When dynamic linking is being used, procedure calls in the source language are translated into instructions that indirectly address the first word of the corresponding linkage block, as shown in Fig. 6-33(a). The translator fills this word with either an invalid address or a special bit pattern that forces a trap.

When a procedure in a different segment is called, the attempt to address the invalid word indirectly causes a trap to the dynamic linker. The linker then finds the character string in the word following the invalid address and searches the user's file directory for a compiled procedure with this name. That procedure is then assigned a virtual address, usually in its own private segment, and this virtual address overwrites the invalid address in the linkage segment, as indicated in Fig. 6-33(b). Next, the instruction causing the linkage fault is reexecuted.

All subsequent references to that procedure will be executed without causing a linkage fault, for the indirect word now contains a valid virtual address. Consequently, the dynamic linker is only invoked the first time a procedure is called and not thereafter.

PROBLEMS

1. Describe the following terms in your own words:
   a. tuning
   b. instruction location counter
   c. associative memory
   d. hash coding
   e. macro
   f. linker
   g. binding time
2. For a certain problem, one percent of the code accounts for 50 percent of the execution time. Compare the following three strategies as to time required to produce the program, plus performance of the program. Assume that it would take 100 man-months to write it in a problem-oriented language and that assembly code takes ten times as long to write and is four times better in terms of speed of the resulting program.
   a. Entire program in problem-oriented language.
   b. Entire program in assembly code.
   c. Entire program in problem-oriented language and then rewrite the one percent.

3. Why are two-pass assemblers used?
4. Do the considerations that suggest two-pass assemblers also hold for compilers
   a. assuming that the compilers produce object modules and not assembly code?
   b. assuming that the compilers produce assembly language for input to an assembler?
5. List some advantages and disadvantages of assembly language compared to a problem-oriented language.
6. List some differences between assembly language and problem-oriented languages.
7. On which pass of the assembly process do the following occur?
   a. Object code generated.
   b. Listing printed.
   c. Literals added to the literal table.
8. Suggest a method to enable assembly language programmers to define synonyms for opcodes. How could this be implemented?
9. Can the following program be assembled in two passes? EQU is a pseudoinstruction that equates the label to the expression in the operands field.

```
A EQU B
B EQU C
C EQU D
D EQU 4
```

10. The Dirtcheap Software Company is planning to produce an assembler for a computer with a 48-bit word. To keep costs down, the project manager, Dr. Scrooge, has decided to limit the length of allowed symbols so that each symbol can fit into a 48-bit word. Scrooge has declared that symbols may consist only of letters, except for the letter Q, which is forbidden (to demonstrate their concern for efficiency to the customers). What is the maximum length of a symbol? Describe your encoding method.
11. Show the steps needed to look up Berkeley, using a binary search on the following list: Ann Arbor, Berkeley, Cambridge, Eugene, Madison, New Haven, New Orleans, Palo Alto, Pasadena, Santa Cruz, Stony Brook, Westwood, Yellow Springs.
12. What is the difference between an instruction and a pseudoinstruction?
13. Make a list of three errors that an assembler should be able to detect in addition to those given in the text.
14. What is the difference between the instruction location counter and the program counter?
15. Is it possible to use binary search on a table whose size is a prime number?
16. Compute the hash codes for the following symbols by adding up the values of the letters (a = 1, b = 2, etc.). The table size is 19.

```
Mercury    Earth
Venus      Mars
```

17. Draw a diagram of the hash table after the following names have been entered in the order given. Use the quadratic quotient method. The table size is 19.

Symbol	Quotient	Remainder
Holland	0	6
Spain	3	1
Germany	1	3
Austria	2	5
Russia	1	1
England	3	11
Scotland	3	3
Kenya	4	4
Tanzania	2	18
Czechoslovakia	3	1
Norway	2	1
Sweden	6	0
Indonesia	3	4

18. Does the quadratic quotient method of resolving collisions eventually probe every table position if
a. the table size is prime?
b. the table size is a power of 2?

19. Show the symbol table after the following statements have been encountered. The first statement is to be assigned to address 1000.

```
EVEREST     L   1,X     (4 bytes)
K2          L   2,Y     (4 bytes)
WHITNEY     MR  1,1     (2 bytes)
MCKINLEY    MR  2,2     (2 bytes)
FUJI        AR  1,2     (2 bytes)
KIBO        ST  1,Z     (4 bytes)
```

20. As single-chip CPUs become cheaper and cheaper, they may be used to form highly parallel machines. Consider a computer with thousands of identical CPUs, all of which can read and write from two shared registers. How could a high-speed associative memory be simulated?

21. Why is macro expansion done on pass one instead of pass two?

22. If a macro definition contains a label, the label will be redefined and an error message will occur every time the macro is called. Devise a method for allowing labels within macros.

23. You are to implement a macro assembler. Your boss has decided that, for esthetic reasons, a macro definition need not precede its first call. How many passes will your assembler need? Discuss.

24. The PDP-11 assembler allows variables to be assigned values during the assembly process. For example, A = 4 means that A is to have the value 4 so that .WORD A assembles a machine word containing the constant 4. Furthermore, A = A + 1 will set A to 5 until it is changed again. What does the following macro do? (Assume that K = 1 just before the macro is called.)

```
.MACRO SURPRISE,X
N = X
.IF NE,X
K = N * K
SURPRISE X - 1
.ENDC
.IF EQ,X
.WORD K
.ENDC
.ENDM
```

25. Most macro assemblers allow conditional assembly on a variety of conditions, but not on the AND or OR of two conditions. Write a macro for some assembler that generates a word of zeros if both of its parameters, A and B, are zero. Write another macro that generates a word of zeros if either or both of its parameters are zero. Now write a third macro that assembles into a word of zeros if and only if one, but not both, of its parameters is zero.

26. Rewrite the macro JUGGLE given in the text to include all possible cases, including interchanging R1 and R2. You may use R3 as a temporary variable.

27. Consider the macro

```
.MACRO HUMBUG,A,B,C
.IF IDN,<A>,<X>
.MACRO M1,P,Q
.WORD P
.ENDM M1
.ENDC
.IF DIF,<A>,<X>
.MACRO M1,P,Q
.WORD Q
.ENDM M1
.ENDC
M1 B,C
.ENDM HUMBUG
```

What is the result of the call: HUMBUG X,4,7?

28. Write a macro PUSH X that pushes its argument onto a stack. Use one of the registers as a stack pointer.

29. Some macro assemblers allow macros to be redefined. Until the redefinition is complete, the old definition is still in force. In particular, if a macro is called inside its own redefinition, the old body is used. Describe the output of the following program (written in PDP-11 notation):

```
.MACRO BAG
.ENDM
.MACRO COLLECT,JUNK
.MACRO BAG
BAG
```

```
.WORD JUNK
.ENDM BAG
.ENDM COLLECT
COLLECT 10
COLLECT 7
COLLECT 4
BAG
```

30. Write a macro in your favorite assembly language that has two parameters, the address of a buffer and the length of the buffer. The macro should generate instructions to zero the buffer.

31. Write a macro GOIF with four parameters that works as shown in these examples.

Macro Call	Meaning
GOIF NY,A,LT,B	IF A < B THEN GO TO NY
GOIF LA,P,EQ,R	IF P = R THEN GO TO LA
GOIF SF,X,GT,Y	IF X > Y THEN GO TO SF

The first parameter is a label, the second and fourth parameters are variables to be compared, and the third parameter is one of the relations: LT, EQ, GT.

32. Which of the following macros can put the assembler into an infinite loop? Give an example of a call that does so.

```
.MACRO M1,A          .MACRO M2,A
.IF EQ,A             .IF EQ,A
M1 A + 1             M2 A
.ENDC                .ENDC
.IF NE,A             .IF NE,A
.WORD A              .WORD A + 1
.ENDC                .ENDC
.ENDM                .ENDM
```

33. Show all the states that the macro expansion stack goes through during the call TAKE2 R,S.

```
.MACRO  TAKE,A,B
INC     A
DEC     B
.ENDM
.MACRO  TAKE2,A,B
TAKE    A,B
TAKE    A,B
.ENDM
```

34. If macro bodies were not allowed to contain macro calls, how would this fact simplify the implementation?

35. Is a one-pass linker possible? If so, what restrictions are needed?

36. A linker reads five modules whose lengths are 200, 800, 600, 500, and 700 words. If they are to be loaded in that order, what are the relation constants?

37. The relocation register used on the Cyber 70 is, in a sense, similar to the base register addressing on the IBM 360. In both cases, the main memory address is the sum of a base register (relocation register) and a constant contained within the instruction. How do you explain the fact that dynamic relocation works on the Cyber 70 but not on the 360 (the 360 does not have virtual memory)? (*Hint:* Who sets the base register?)

38. Prof. Martin Metabit has made the following proposal: each register on a machine with general registers used for base registers and other functions should be equipped with an extra bit that could be set or cleared by special instructions. Could this proposal solve the dynamic relocation problem? Does the order of setting the bit and loading the base register matter?

39. A group of overeager freshman electrical engineering students heard about the dynamic relocation problem on their IBM 360 (without paging) and crept into the computer center late one night to rewire the machine so as to make register 15 always contain the program counter (like register 7 on the PDP-11). Does this change solve the dynamic relocation problem by allowing self-relative addressing as on the PDP-11? (*Hint*: Consider the displacement size and address space size on the two machines.)

40. Cyber 70 instructions may be either 15 or 30 bits. An instruction may not be split over two words. Only 30-bit instructions contain memory addresses. How many relocation bits are needed per word?

41. Does base register addressing eliminate the need for the linker to relocate memory reference instructions? You may assume that the standard calling sequence always loads a base register with the address of the start of the procedure. Does anything else have to be relocated? (*Hint:* Think about address constants—words containing a pointer to a labeled statement.)

42. Write an assembly language procedure to store and retrieve symbols of up to eight characters, using hash coding and the quadratic quotient method.

43. Write an assembler for the computer for the machine of Fig. 4-9. The assembler should handle
    a. the 12 mnemonics given in the figure (PUSH, POP, etc.).
    b. symbolic labels up to eight characters.
    c. the pseudoinstructions
       BEGIN (start of an assembly)
       END (end of an assembly)
       CON (assemble a one-word decimal constant)
    d. literals (e.g., PUSH =1).
    e. comment cards with a * in column 1.
    The assembler should print a listing, giving the assembled code for each instruction in octal or hexadecimal, whichever you think is more esthetically pleasing.

44. Add a simple macro facility to the assembler of Problem 43. Macros may have parameters, but a macro may not be called from inside a macro body, and a macro may not be defined inside a macro body. Devise and implement a method for handling labels that appear inside macros so that they will not be multiply-defined if the macro is called twice.

7 MULTILEVEL MACHINES

The preceding four chapters each covered one specific level found on many modern computers. In this chapter we will take a more global view and examine some of the techniques and problems associated with organizing a computer in a structured way, as a hierarchy of levels. The topics to be covered include methods for implementing new levels, the order of the design decisions, the use of multilevel machines for producing portable software, self-virtualizing machines (machines in which the same level occurs more than once), and the nature of the principal interface level.

7.1. METHODS OF IMPLEMENTING NEW LEVELS

Three different methods for implementing new levels will be summarized in the following sections—interpretation, translation, and procedural extension. Interpretation and translation are the most important methods, with procedural extension only being used for certain specialized applications.

7.1.1. Interpretation

The essence of interpretation is that a process fetches, examines, and carries out the instructions contained in some program. The program being interpreted is really the data used by the interpreter, just as a file of employee names and salaries is the data for a payroll program. The process carrying out the interpretation acts exactly like a "hardware" processor, and the program being interpreted cannot tell whether it is being interpreted by a lower-level process or whether it is running on a real hard-

ware processor of equivalent speed. In both cases, the process whose program is being interpreted passes through exactly the same sequence of states.

It is possible to have a hierarchy of interpreters, each one interpreting the level above it. As an example, consider a PL/I program that interprets PDP-11 conventional machine level instructions. This program can be used on an IBM 370 by first having the 370's PL/I compiler generate a 370 level 3 program equivalent to the PL/I program. This step is called compilation of the PL/I program.

The resulting program consists of load register, store register, and other "ordinary" instructions, as well as the 370's file i/o OSML instructions for performing i/o (see Section 5.2). The file i/o instructions are needed to simulate the PDP-11 when it reads from its disk, for example. The number of interpreters active depends on whether an "ordinary" or an OSML instruction is being executed by the level 3 program. For an ordinary instruction, there are two interpreters, the microprogram and the PDP-11 simulator. For an OSML instruction, such as reading a record from a file, three interpreters are at work: the microprogram, the operating system, and the PDP-11 simulator.

Figure 7-1 shows a five-level machine. After compilation has been completed (i.e., during execution), four processes will be running, corresponding to the execution of the microprogram, the operating system, the PDP-11 simulator, and the PDP-11 program running on the simulator. Each process has its own program, its own state

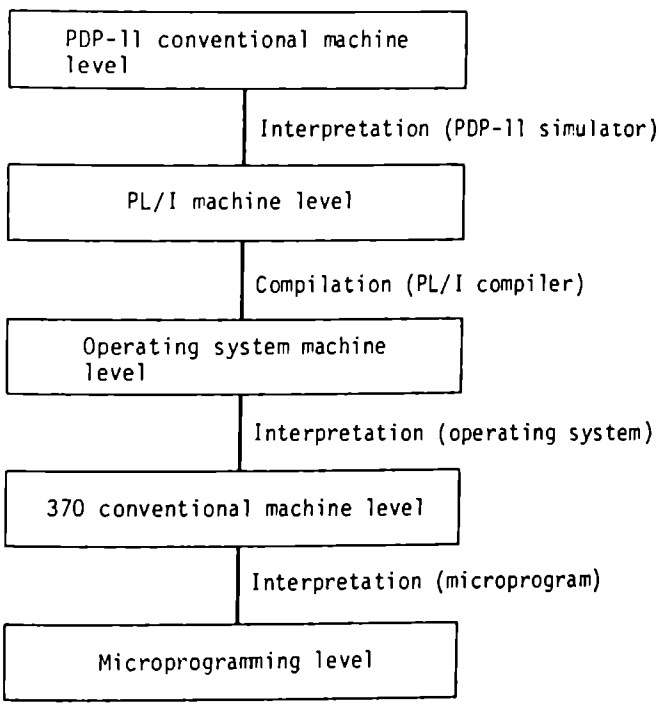


Fig. 7-1. A PDP-11 simulator written in PL/I for the IBM 370 viewed as a 5 level machine.

vector, and its own program counter. When the PDP-11 simulator executes an "ordinary" instruction, the operating system becomes inactive, however.

Every microinstruction advances the operating system through part of one of its instructions. Every time the microprogram does so, a piece of the OSML instruction that it is working on is carried out. Every time the PDP-11 simulator gets through one of its instructions, which will require many microinstructions, it advances the PDP-11 process part of the way through its next instruction. In other words, every microinstruction contributes to the progress of not just one but four processes, all running simultaneously on different virtual machines.

7.1.2. Translation

A language, L, implicitly defines a virtual machine—namely, the virtual machine whose machine language is L. This fact is equally true for problem-oriented languages, such as PL/I, as for the conventional machine level type. In principle, there is no reason why a computer could not be provided with a PL/I virtual machine, a level whose machine language is PL/I. Such a machine could directly execute PL/I programs, just as the 370 conventional machine level can directly execute programs written using the instructions of Fig. 3-4. All that is required is a PL/I interpreter. Such an interpreter could be written for the level 3 machine, for example, thus making the PL/I machine level 4.

Although a PL/I machine is certainly conceivable, it is rarely implemented as such for a very good reason. The syntaxes of the microprogramming level and conventional machine levels are basically very simple: an opcode field and a few address fields. On the 370 conventional machine level, for example, the leftmost eight bits form the opcode. As a result, decoding the instructions is easy for the microprogram. A 256-entry table, with one entry per opcode, can provide the address of the part of the interpreter to be used.

Instruction decoding for a PL/I machine is far more difficult. Determining what operations are needed to execute even a simple PL/I statement like

```
IF ANDY = HUNGRY THEN COOKIE_COUNT = COOKIE_COUNT - 1;
```

will require considerably more time than simply using an eight bit number as an index into a table, because manipulating variable-length character strings is inherently more complicated than performing one indexing operation. Worse yet, each time a PL/I statement is to be executed, its syntax must be analyzed again, even though it had already been analyzed when previously used. If a statement in a loop is executed 1000 times, its syntax will be analyzed 1000 times, which is obviously very inefficient.

Translation provides a method for efficiently implementing a level with an elaborate syntax. The translator first reads and analyzes the program to be executed. Then it produces as output a new program (the object program) in a different language (the target language) that is equivalent to the original. Since one of the main reasons for translation is to avoid having to perform long and time-consuming syntax analysis repeatedly, languages used as target languages always have instructions with a simple syntax, such as an opcode field and a few address fields.

The most important kinds of translators are assemblers, compilers, and macro processors. Assemblers translate languages that are, in fact, symbolic representations of the target languages. The assembler's principal task is replacing the symbolic names of the variables and the symbolic opcodes by binary numbers, in order to spare the interpreter the need for time-consuming character handling.

Compilers translate problem-oriented languages to a form suitable for interpretation or, in some cases, to an assembly language for subsequent translation by an assembler. The distinction between a compiler and an assembler is more a matter of degree than anything else. If a great deal of syntax analysis is to be performed, the translator is called a compiler; otherwise it is called an assembler.

General-Purpose Macro Processors

Macro processors represent a third type of translator. The assembly language macros of Section 6.4 represent a very simple form of macro. Far more general and powerful macro processors are available, however, than those provided by most assemblers. Basically, a general-purpose macro processor first reads and stores a series of macro definitions. Then it reads the program to be translated, looking for macro calls and replacing them by the previously defined bodies.

The resulting text, which may be in any language, is written onto a file rather than being fed directly into an assembler. This output file can later be used as input to any assembler or compiler. Macro processors simply recognize certain character strings (the macro calls) and replace them by other character strings (the macro bodies).

ML/I (Brown, 1967) is an example of a general-purpose macro processor. ML/I allows the programmer to define his own syntax by defining patterns that ML/I is to recognize and replace by other patterns. As an example of a user-defined macro, we will show how statements of the form

```
IF PROGRAM = PERFECT THEN STUDENT = HAPPY;
IF TIME = NOON THEN STOMACH = EMPTY;
IF MONTH = AUGUST THEN BEACH = FULL;
```

can be handled by ML/I and converted into PDP-11 assembly language, for subsequent use by the PDP-11 assembler.

These statements all have the form

```
IF * = * THEN * = *;
```

where * indicates that an actual parameter is expected at that position. An ML/I macro to convert this kind of statement to PDP-11 assembly language is

```
MCDEF IF     =     THEN     =     ;
AS<    CMP   %A1.,%A2.
       BNE   .+8
       MOV   %A4.,%A3.
>
```

The MCDEF statement is a MaCro DEFinition for a macro with five delimiters:

1. IF
2. =
3. THEN
4. =
5. ;

A call of this macro must supply an actual parameter between each of the four pairs of delimiters. In the call

```
IF NAME = FRODO THEN TYPE = HOBBIT;
```

the actual parameters are

actual parameter 1 = NAME
actual parameter 2 = FRODO
actual parameter 3 = TYPE
actual parameter 4 = HOBBIT

The macro body is enclosed between < and >. The marker %A1. is replaced during expansion by actual parameter 1, the marker %A2. is replaced during expansion by actual parameter 2, and so on. ML/I numbers its formal parameters rather than naming them because it is possible to define macros with a variable number of parameters. For example, a single CALL macro could be defined so that it could be used with 0, 1, 2, or arbitrarily many actual parameters. The preceding macro call is replaced by

```
CMP  NAME,FRODO     (compare NAME to FRODO)
BNE  .+8            (if unequal, skip 1 statement)
MOV  HOBBIT,TYPE    (move HOBBIT to TYPE)
```

ML/I allows macros far more complicated than those shown in this example. A macro may have arbitrarily many alternative sets of delimiters, a feature that could be exploited to allow the IF macro above to handle <, >, and so on, as well as =. The programmer can write conditional macros whose expansion depends on both the actual parameters and the delimiters—for example, checking for = or <. The pattern to be recognized as a macro call may be arbitrarily long and may have an arbitrary number of parameters. Furthermore, the actual parameters themselves may be arbitrarily long and may even contain nested macro calls.

A general IF macro could be written to handle THEN clauses several pages long, if need be, including other calls of the IF macro. Macros are fully recursive, and a macro may be called from within its body as well as from within one of its actual parameters. Macros may be redefined or deleted. Provision is made for performing both arithmetic tests and string comparisons during the macro expansion process, the

results of which can direct the course of the expansion. In short, ML/I can be used to translate a fairly sophisticated programming language with an arbitrary syntax.

7.1.3. Procedural Extension

It is common for applications packages in statistics, computer graphics, and other areas to be implemented as a collection of procedures in PL/I, FORTRAN, or another language. As an illustration of this approach, consider the following representative procedures from a statistics package.

```
CALL READ(X);       /* READ A DATA DECK INTO THE VECTOR X */
CALL READ(Y);       /* READ A DATA DECK INTO THE VECTOR Y */
CALL CORRELATE(X,Y,R);          /* CORRELATION COEFFICIENT */
CALL PRINT(R);                  /* PRINT R */
CALL LEAST_SQUARES(X,Y,A,B); /* FIT A STRAIGHT LINE TO THE DATA */
CALL PLOT(X,Y);                 /* PLOT THE DATA */
CALL DRAW_LINE(A,B);            /* PLOT THE BEST FIT LINE */
CALL NEXT_FRAME;       /* PREPARE TO MAKE ANOTHER PLOT */
```

Although this language is technically PL/I, an unsophisticated user who knew nothing about PL/I might think of it as a programming language with instructions READ, CORRELATE, PRINT, LEAST_SQUARES, PLOT, DRAW_LINE, NEXT_FRAME, and various other similar instructions. He can combine these one-line "instructions" to form programs, just as an assembly language programmer can combine other one-line instructions to form programs.

In one sense, these instructions define a virtual machine. This virtual machine can also be implemented by translation or interpretation, of course. In another sense, this level is no different than the PL/I level. Construction of a level using procedures is not as widely used as the other two methods.

7.2. DESIGN STRATEGIES FOR MULTILEVEL MACHINES

There are three conceptually different methods by which a multilevel system can be designed. Method 1, the **top-down** design technique, consists of first designing the topmost level, the one that the end user ultimately sees. Next, the level below it is designed, then the level below that, and so forth, until a level is reached that can be interpreted by an interpreter running on the bare hardware. In the top-down method, the levels are designed starting at the top and working downward.

Method 2, the **bottom-up** design technique, is precisely the opposite of the top-down approach. When designing bottom up, the level closest to the hardware is designed first, then the level above it, and so on, until the top level is reached.

Method 3, the **middle-out** design technique, consists of first designing one of the intermediate levels and then working upward and downward simultaneously. This approach has the advantage that once one of the middle levels has been completely specified, one design group can get to work designing upward, starting at that level,

while at the same time another design group can get to work designing downward from that level, toward the hardware.

7.2.1. Top-Down Design

The top-down design philosophy is intuitively appealing, since a designer using it starts out by carefully specifying the virtual machine (and language) that the customer will ultimately use. He must specify the data types to be provided and the instructions available for manipulating them. Sometimes this step is done even before the project begins.

For example, if the goal of the project is to design and implement a PL/I or an ALGOL 68 level, the designer need not first design PL/I or ALGOL 68, for they have already been carefully specified. If, on the other hand, the project is to design a level (machine and language) to be used for research in artificial intelligence, then the choice of features to include at the top level is an open question that must be first resolved.

After the top level has been completely and precisely specified, the designers try to think of another level from which the implementation of the top level would be convenient. In other words, they must ask themselves: "If we were to write an interpreter for the top level, in what language would we like to write the interpreter?" Or they might ask: "If we were to write a compiler to compile programs written for the top level into some other language, into what language would we like to compile them?"

As an example of the top-down approach, consider the design of a computer system to be used as a base for an automated library. The first step is to decide what kind of features and services the library will have. Will it simply have the books and magazines and an automated subject, title, and author catalog? Will it be able to accept and provide responses to such requests as "Give me a list of all journal articles in 1976 about the mating habits of lemmings?" This level can be thought of as a virtual library machine, as shown in Fig. 7-2(a).

Once the virtual library machine has been specified, the designers must devise a method for implementing it. One technique would be to invent a programming language that is particularly suited to string and character manipulation. Alternately, they may decide that an existing string-processing language, such as SNOBOL4, is adequate. The system designed so far is shown in Fig. 7-2(b).

The designers must now face the problem of implementing the virtual machine defined by the string-processing language SNOBOL4. One method would be to invent a more or less conventional computer that had, in addition to the usual kinds of instructions, special instructions for processing strings. Such instructions might include instructions for moving a string, comparing two strings, concatenating two strings, searching a string for a substring, and appropriate string i/o. Programs written in the string-processing language would first be compiled to this augmented conventional machine level and then run at that level. Figure 7-2(c) illustrates the design up to this point.

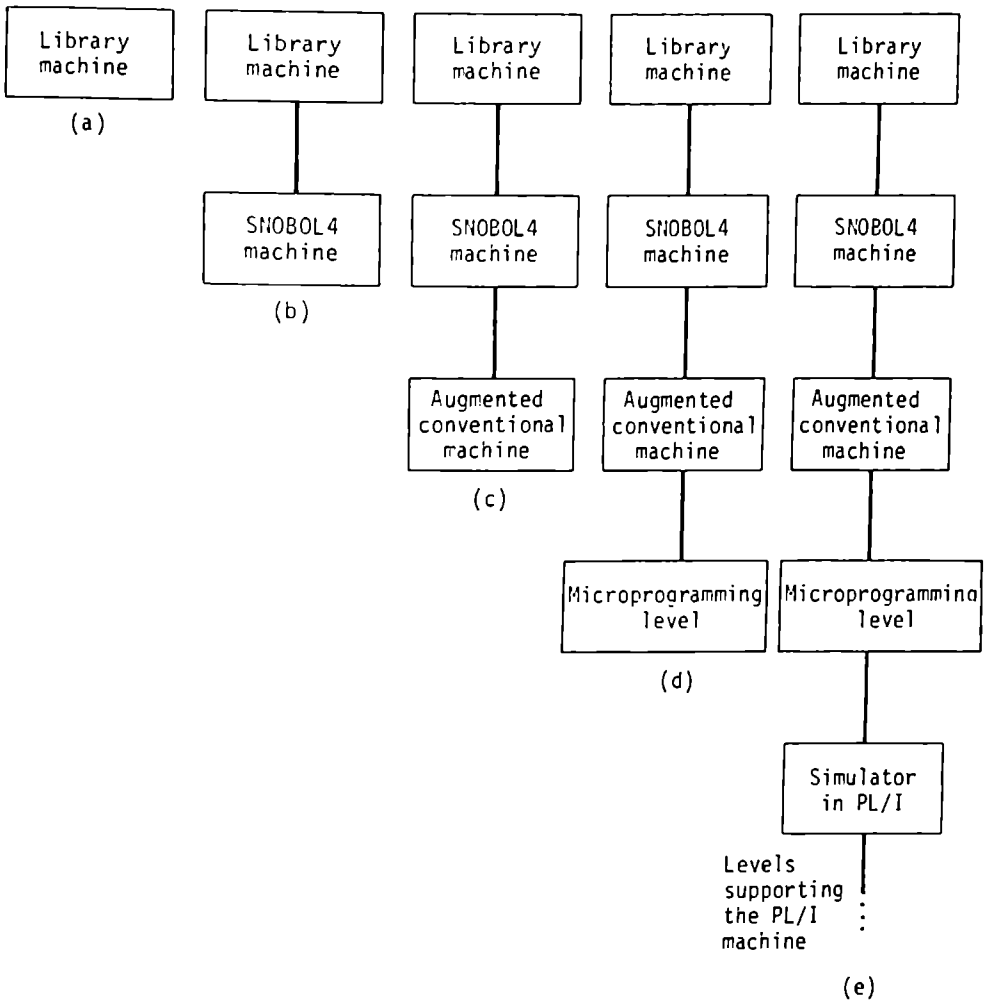


Fig. 7-2. Steps in the top down design of a virtual library machine.

The next step in the design is to find a method for supporting the augmented conventional machine level. An obvious choice is to buy or build a microprogrammable CPU to interpret it. The microprogramming level of the machine chosen should have good character-handling facilities. Figure 7-2(d) shows the complete system.

Before investing time, energy, and money in the purchase or construction of the CPU, it would be useful to be able to measure the performance of the complete system. Otherwise, when faced with the choice of using the low-cost Turtle-brand computer or the expensive Jackrabbit-brand machine, there will be no basis for making intelligent decisions. The way to handle this situation is to write a simulator (interpreter) for the microprogramming level before actually building or buying anything. Figure 7-2(e) shows the implementation of the microprogramming level as a PL/I program. Although not shown, quite a few levels may be beneath the PL/I level,

supporting it. Using the simulator, the system can be constructed and partially tested to see how well it performs. If it performs badly, the design can be changed before the (more or less irreversible) decision to buy some particular hardware is made.

7.2.2. Bottom-Up Design

The opposite approach to the top-down design is to begin with the existing hardware and construct a level above it whose properties are more attractive (or perhaps less obnoxious) than those of the hardware. This new level is then used as a base for creating a still more convenient virtual machine. As many new levels as necessary are added until the desired virtual machine is achieved.

Many advanced operating systems are constructed not as a single level, as suggested in Chapter 5, but as a hierarchy of levels. Dijkstra's (1968b) "THE" system is the spiritual progenitor of this type of operating system.

These designs usually begin at the conventional machine level rather than at the microprogramming level. Most conventional machine levels have an interrupt system, which means that any instruction may be followed in time either by its normal successor or by the first instruction of some interrupt procedure, should an interrupt occur.

Consequently, the precise sequence of states that occurs depends on the exact time that the interrupts happen. Since the physical motion of an i/o device is rarely reproducible to milliseconds, let alone microseconds or nanoseconds, if a program is rerun several times, it is highly unlikely that exactly the same sequence of states will occur on each run. A process whose behavior is not uniquely determined by its program, data, and initial state is said to be **nondeterministic**, to distinguish it from the normal deterministic or sequential process.

Since programming for a deterministic machine is much easier than programming for a nondeterministic machine, the level above the conventional machine level is chosen to be deterministic. Instead of having a single nondeterministic process, this level has several cooperating deterministic processes running in "parallel." The only process running on the conventional machine level is the interrupt handler.

When an interrupt occurs, the interrupt handler stops the process currently running and starts the appropriate interrupt process. After the interrupt process has done its job, it signals that it is finished, and the other process continues. Each of these processes is completely deterministic. The only peculiarity is that sometimes the interval between two instructions is very long because another process ran for awhile. However, in order for a process to be deterministic, the only feature that need be guaranteed is that the sequence of states be fully reproducible. There are no guarantees about how much or how little time may elapse between instructions. The level on which all processes are deterministic will be called the deterministic virtual machine level, as shown in Fig. 7-3(b).

Programs running on the deterministic machine level reference physical memory locations. Since virtual memories provide programmers with the convenience of a large and, in some cases, segmented address space, most modern computers have a

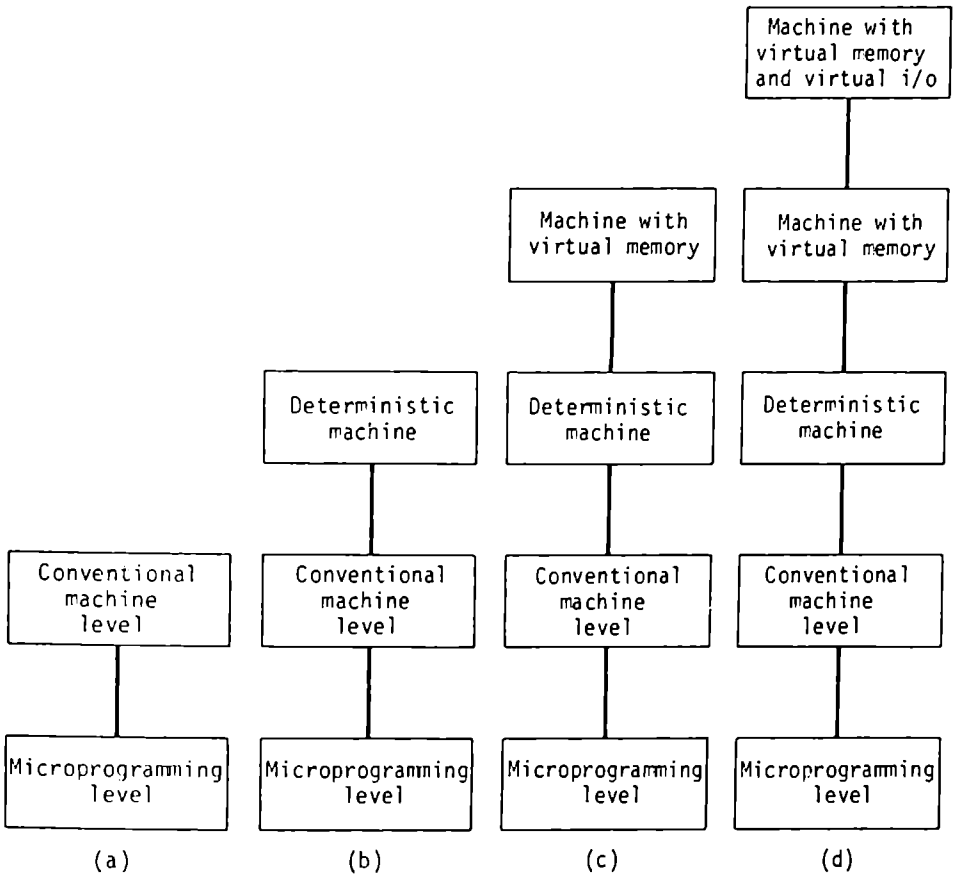


Fig. 7-3. Steps in the bottom up design of a multilevel operating system.

virtual memory. The process or processes for supporting the virtual memory can run on the deterministic machine. This situation is shown in Fig. 7-3(c). The level above the deterministic machine level has the following two properties:

1. All processes are strictly deterministic.
2. Processes may use the virtual memory without worrying about its implementation.

Note that not every instruction executed by a process running on the virtual memory machine is interpreted by a process on the level below. On the contrary, most instructions are interpreted directly by the microprogram, and only those instructions that cause page, segment, or protection faults cause processes at the deterministic machine level to become active.

Programs running on the virtual memory machine must directly access each i/o device by giving its hardware device number. It is more convenient for the programmer to have an instruction TYPE LINE that simply types a line on his terminal. Accord-

ingly, the next level provides virtual i/o instructions, as shown in Fig. 7-3(d). When a process on the virtual i/o machine executes a TYPE LINE instruction, a process on the machine below it carries it out.

7.2.3. Middle-Out Design

In practice, designs are rarely pure top down or bottom up. A pure top-down design has the inherent danger that some attractive features built into the uppermost virtual machine may be difficult, if not impossible, to implement on the available hardware. A pure bottom-up design may proceed through many levels before the designer discovers that the resulting machine is not what the customer requested.

Good designers generally have some idea of what the ultimate top and bottom levels will be like, and they (consciously or unconsciously) use this information during the design process. An experienced designer doing a top-down design is not going to include any features in the top level unless he is confident that the level can be implemented somehow, even if he does not have the details worked out yet. As a result, the design is not really pure top down, since information about the bottom level has influenced the design of the top level.

Many designers start in the middle and then work outward in both directions. Many computers have been designed middle out, with the conventional machine level being specified first. After that level was specified, it was possible to begin the detailed specification of the microprogramming levels and microprograms, on one hand, and the operating systems and compilers on the other. Middle-out design allows the upward and downward designs to proceed simultaneously if two design groups are available.

7.3. PROGRAM PORTABILITY

Ever since the appearance of the second computer, several decades ago, people have been trying to devise ways to allow programs written for one computer to be run on another computer. When a university or corporate computer center replaces its old computer with a shiny new one, the users want to run their old programs on the new machine without having to rewrite them. When a student graduates and goes to work, he may want to bring some programs with him. When a research worker discovers that one of his colleagues elsewhere has written a program that he needs, he would like to use it, even if the program was written for a brand X computer and his computer is a brand Y. A program that can be moved from one computer to another with relatively little effort is said to be **portable.**

Many proposals have been made over the years to increase program portability. In the following sections we will briefly examine six of these methods:

1. Universal programming language.
2. Brute force approach.

3. UNCOL.
4. Do-it-yourself virtual machines.
5. Emulation.
6. Networks.

Most of these techniques involve multilevel machines in one way or another.

7.3.1. A Universal Programming Language

The first approach to achieving portability is to design a general-purpose programming language and then request or require all programmers to use it exclusively. By implementing the virtual machine defined by this language on a large number of computers (by providing compilers or interpreters for all of them), programs can be moved easily from computer to computer. This approach is shown in Fig. 7-4.

All attempts to produce such a universal programming language have failed miserably with respect to their goal of universal acceptance and use. On the contrary, hundreds of programming languages are already in use, and their numbers are increasing like rabbits.

The basic reason for the lack of success of a universal programming language is that each programmer wants a language particularly suited to his application and does not want to pay (in terms of a large, inefficient compiler and difficult-to-learn language) for facilities that he does not need or want. Systems programmers use binary trees heavily, physicists need real matrices, librarians require character strings, and administrators want to be able to handle very large data bases. Each group naturally prefers a language that emphasizes the data types and operations needed for its application, rather than a general-purpose language.

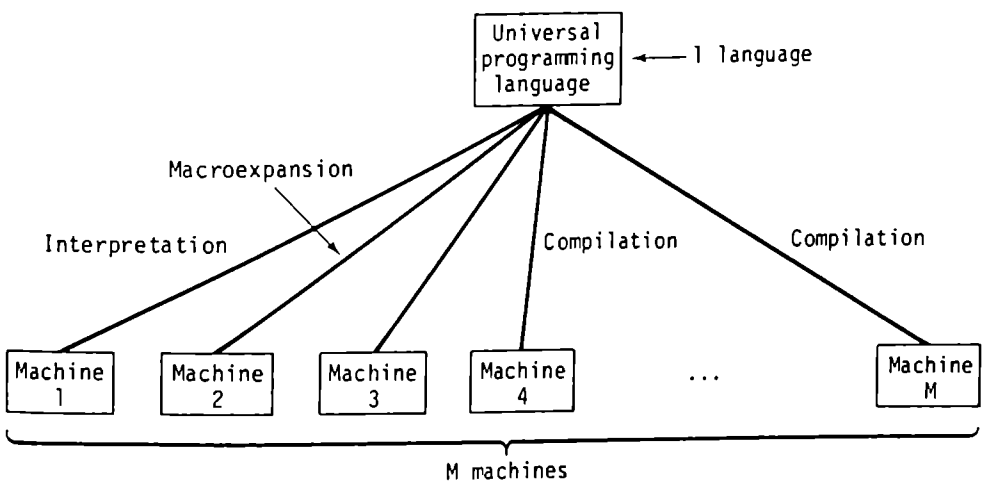


Fig. 7-4. A universal programming language available on all computers.

Although no universal programming language is used by everybody for everything, two languages are very widely used: COBOL and FORTRAN. You might think that a FORTRAN program written for one computer will run the first time it is tried on another, but such is rarely the case. Some of the difficulties are listed below.

1. Almost no manufacturer's compiler adheres exactly to the official ANSI Standard. Nearly all have "new, improved" (and incompatible) extra features.
2. Word lengths vary from machine to machine, which means that the largest integer available also varies.
3. The smallest positive floating-point number is machine dependent. This has drastic effects on programs that iterate until something is "less than epsilon."
4. The number and type of i/o devices available differ from machine to machine. An eight-tape sort will have trouble on a computer with only five-tape drives.
5. Subroutine libraries differ.
6. The level 3 virtual instructions depend on the operating system. If a program creates a file of 136 character records, it will not work on a computer whose operating system insists that all records be less than or equal to 132 characters.
7. The program may not fit in the new machine's memory.
8. The character sets may be incompatible. A certain bit pattern may be a printable character on one computer and illegal or an end-of-record mark on another.
9. One's complement computers may give different results than two's complement computers. The biggest source of problems is tests on -0.
10. If the program contains any assembly language subroutines, they will have to be rewritten.

Most of these problems have the same root cause: the virtual machine defined by the programming language is not completely specified. Details like word lengths, i/o, and libraries are not part of the definition.

7.3.2. The Brute Force Approach

Once the goal of a universal programming language has been abandoned as hopeless, a second approach is to provide an implementation of every language on every computer. In this way, no matter what language a programmer chooses to use, the virtual machine defined by it will be implemented on whatever real compiler he has available.

If there are L programming languages and M machines, the number of implementations (compilers and interpreters) needed is the product $L \times M$, as shown in Fig. 7-5. Hundreds of different computers have been built and even more languages have been developed, so the number of implementations needed probably exceeds 100,000. Producing that many compilers and interpreters would be a massive undertaking,

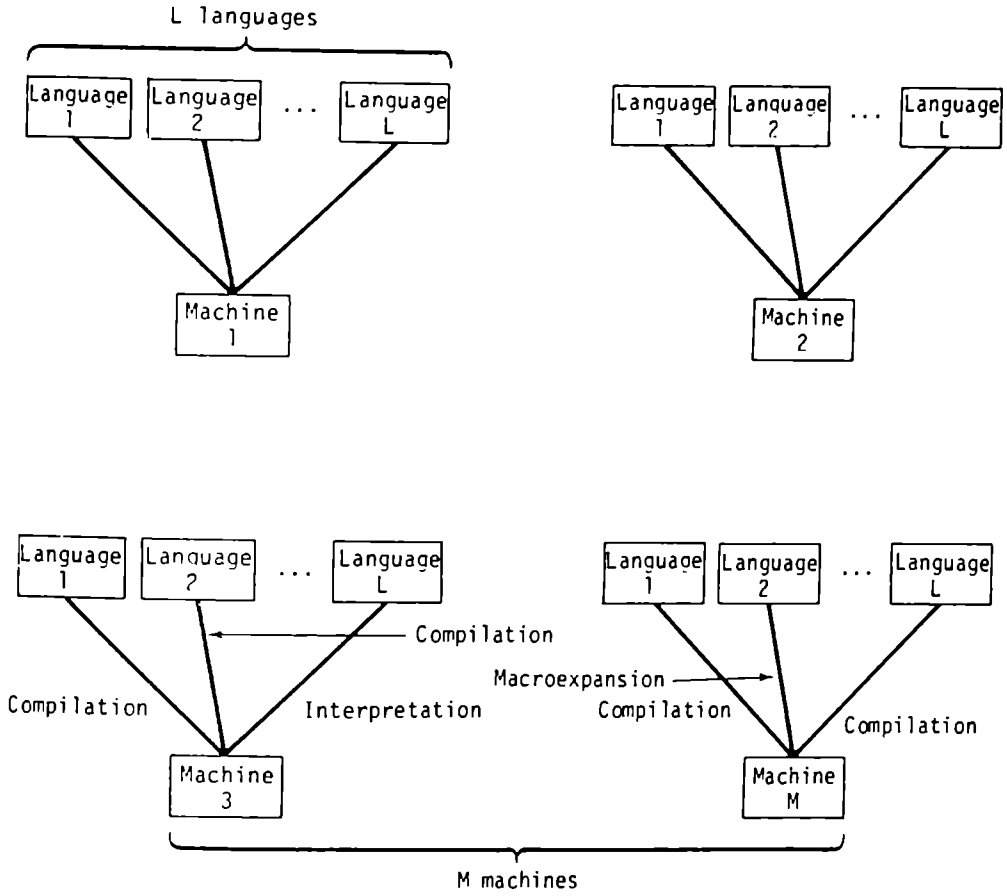


Fig. 7-5. A separate implementation of each language on each machine.

far exceeding the available manpower. Moreover, it would be a never-ending task, since new languages and computers are constantly appearing.

7.3.3. UNCOL

A third attempt at solving the portability problem is to design a virtual machine to be used as a base for implementing various language levels. To achieve the goal of portability, this virtual machine, called an UNCOL (level) (UNiversal Computer Oriented Language), should be implemented on as many computers as possible. The idea of an UNCOL is illustrated in Fig. 7-6. The UNCOL level defines a language in which it is possible to write interpreters for PL/I, FORTRAN, SNOBOL4, PASCAL, and so on, or in which to write compilers that translate programs in these languages into UNCOL programs. The UNCOL approach is an application of multilevel machines.

In order to use these compilers and interpreters, it is necessary to support the UNCOL level on each existing computer. The UNCOL level can be implemented

by equipping each computer with either an UNCOL interpreter or an UNCOL-to-machine language translator. The UNCOL strategy requires only L + M compilers and interpreters, as shown in Fig. 7-6, compared to the L × M compilers and interpreters needed by the brute force approach of Fig. 7-5.

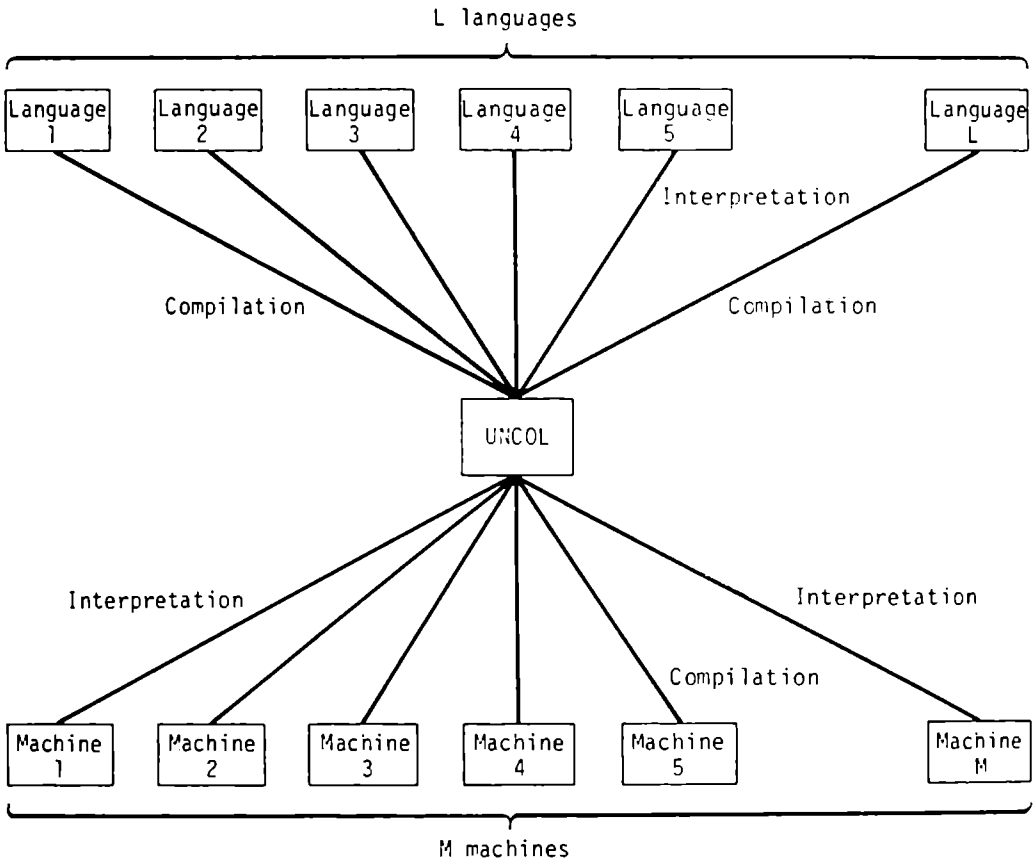


Fig. 7-6. Implementation of L languages on M machines using an UNCOL.

The UNCOL approach to portability has been adopted by IBM, DEC, and other manufacturers. IBM, for example, manufactures a whole series of different computers, which we have blithely lumped together under the name "370." Each of these computers is microprogrammed. Each one has a microprogramming level (i.e., the actual hardware of the computer) that is completely and totally different from all the others. Some of the ways in which computers can differ are

1. Instruction length.
2. Bus widths (e.g., ALU output width).
3. Number and size of the memories.

Model	CPU	ALU width	Microinstruction length
360/30	2030	8	50
360/40	2040	8	56
360/50	2050	32	88
370/115	3115	8	20
370/125	3125	16	19
370/135	3135	16	16
370/145	3145	32	32
370/158	3158	32	72
370/168	3168	64	108

Fig. 7-7. Comparison of some models of the 360 and 370 series.

4. Instruction set.
5. Number and function of processor registers.
6. Number and organization of the i/o processors.

In terms of these measures, no two models of the 360 or 370 series are even similar. Figure 7-7 shows the ALU width and microinstruction length for a few models of the 360 and 370 series as examples. Obviously the 3125 with a 19-bit microinstruction and 16-bit ALU width has little, if any, relation to the 3168 with its 108-bit microinstructions and 64-bit ALU width. The other aspects of the 370 CPUs are as different as these. The various models of the 370 series are simply different computers, just as the 370/125 and PDP-11/45 are different computers.

IBM is faced with the problem of customers whose work load has gradually increased over the years and who want to trade in their current 370 model for one of the larger and more powerful ones. If a 370/125 customer did all his programming in 3125 machine language, he would be stuck. There would be no way of moving him up to a 370/135, short of rewriting everything.

IBM's solution to this problem is to use the 370 conventional machine level as an UNCOL. Each model of the 370 series has an interpreter for interpreting 370 conventional machine level programs. Since each interpreter supports exactly the same virtual machine, programs can be easily moved from one model to another despite the enormous differences in the hardware.

To complete the UNCOL picture, IBM has written compilers that translate PL/I, FORTRAN, COBOL, and other languages to the UNCOL level, as shown in Fig. 7-8. Since the UNCOL is available on all models, only one PL/I compiler is needed for all of them despite their lack of similarity. Strictly speaking, the compilers produce level 3 and not level 2 programs as output, but since the operating systems are also portable (they all run on the same level 2 machine), the level 3 machine is also an UNCOL.

The UNCOL approach has enormous benefits to IBM compared with the brute force approach. When IBM wants to introduce a new computer, it need only provide one program for it: an interpreter for the 370 conventional machine level. Writing one interpreter is obviously less work than writing compilers for PL/I, FORTRAN,

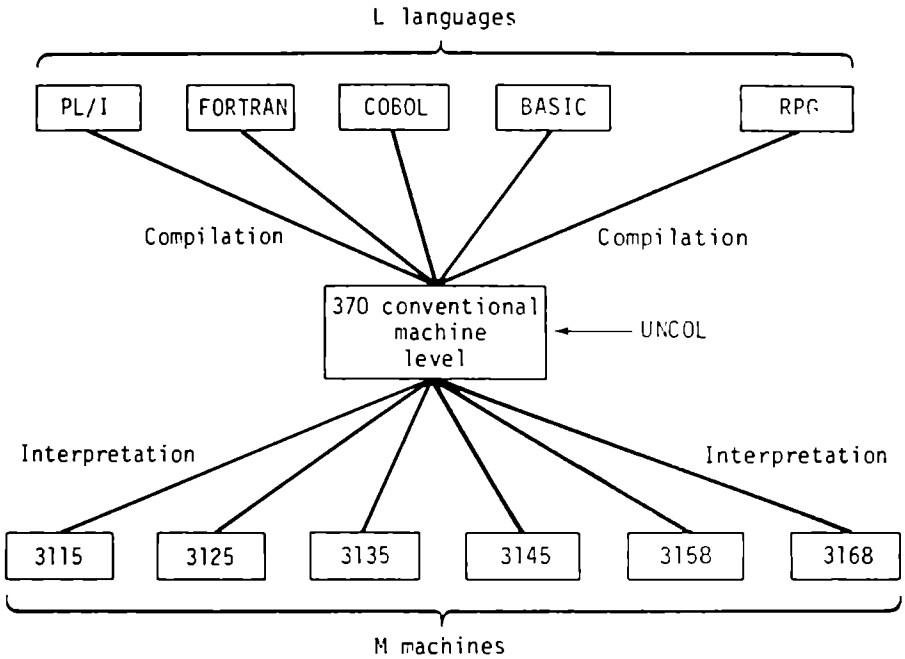


Fig. 7-8. The 370 conventional machine level as an UNCOL.

COBOL, and the rest. Because only one program is needed, it is economically feasible to have a team of the very best programmers spend months carefully optimizing every instruction in the interpreter.

Nevertheless, the strategy of using an UNCOL to achieve portability is not without its price. Bell and Newell (1971) estimate that a given algorithm programmed in the 360 conventional machine language and interpreted by a microprogram running on the 2025 CPU (used by the 360/25) could be speeded up by a factor of 45 if the conventional machine level were skipped and the algorithm were programmed directly in 2025 machine language. They also state that a 2025 microprogram interpreting a FORTRAN virtual machine might easily approach the speed of FORTRAN running on the 360/50 conventional machine level. Since they give the average cost of the 360/50 CPU and memory as roughly a factor of 6 more than that of a 360/25, the cost of portability to the user is not negligible.

7.3.4. Do-It-Yourself Virtual Machines

Although the UNCOL strategy solves the problem of moving programs from one model of a given series (370, PDP-11, etc.) to another, it does not solve the problem of moving programs between machines built by different manufacturers. However, multilevel machines can again be used to solve this problem to a certain extent.

A programmer who wanted to produce a piece of portable software could proceed as follows. First, he could design a virtual machine, V, on which his program is to

run. This virtual machine must satisfy two criteria:

1. It must be suitable for the program to be written.
2. It must be simple enough to be easy to implement on existing computers.

Having designed a virtual machine, he can then proceed to write his program in its machine language.

In order to run this program on an existing computer, C, the virtual machine V must first be implemented. Because V was chosen for easy implementation, this task is relatively straightforward. The most common method for implementing virtual machines of this kind is by macro expansion. For each instruction in V, the implementor writes a macro that expands into C's assembly language. The result of the macro expansion is a program in the assembly language of the computer on which the portable software is to run. This situation is illustrated in Fig. 7-9. The method differs from the UNCOL method in that each portable program can use a different virtual machine, if need be.

To aid subsequent implementors of V, the designer of V may provide a diagnostic program for V, similar to the hardware diagnostic programs used by maintenance personnel to check out real hardware. Such a diagnostic program could add 1 and 1 and then compare the result to 2. If the result were not 2, it might print out the message

THE VIRTUAL ADD INSTRUCTION IS NOT WORKING PROPERLY

A well-designed diagnostic program can make it possible to write and debug the macros for a simple virtual machine in less than a week.

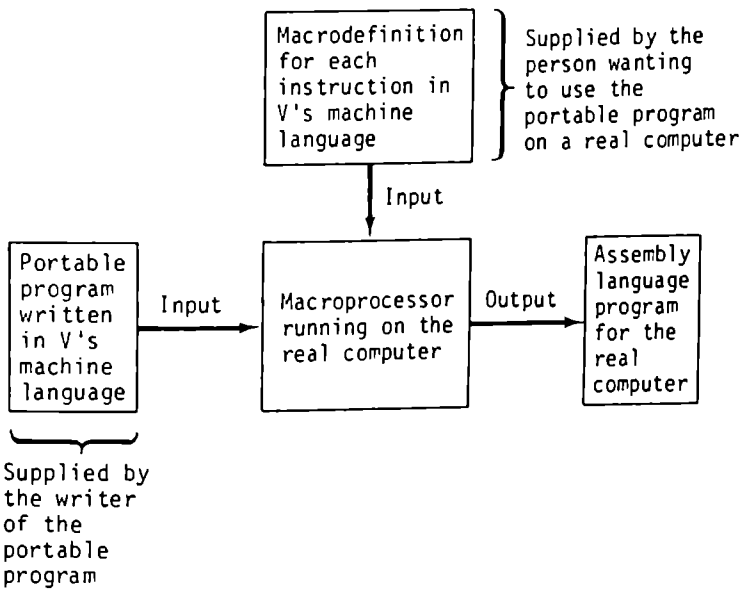

Macrodefinition for each instruction in V's machine language
Supplied by the person wanting to use the portable program on a real computer
Input
Portable program written in V's machine language
Input
Macroprocessor running on the real computer
Output
Assembly language program for the real computer
Supplied by the writer of the portable program

Fig. 7-9. Use of a virtual machine, V, for producing portable software.

The macro processor used to implement the virtual machine can be either the assembler's macro processor or a general-purpose macro processor, such as ML/I. The general-purpose macro processors have the advantage of being able to handle a wide variety of input languages, besides being very powerful programming languages themselves. The writer of the portable software is thus provided with more flexibility and convenience in designing his virtual machines.

ML/I-type macro processors have the disadvantage that, before they can be used, they themselves must be implemented on the various target machines. However, the designers of these macro processors usually provide a portable version, written for a very simple virtual machine, V1. To implement ML/I on a new computer, one must first write macros for the instructions in V1 and use the assembler's macro processor to translate ML/I to machine language. Once ML/I is running, it can be used to implement the virtual machines used by the portable programs. After the step shown in Fig. 7-10 has been performed, the ML/I macro processor is available for translating other virtual machines, as shown in Fig. 7-9.

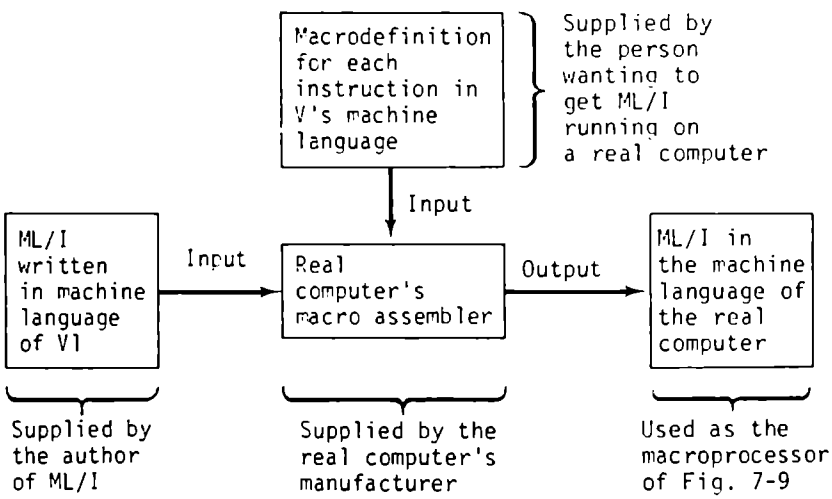

Macrodefinition for each instruction in V's machine language
Supplied by the person wanting to get ML/I running on a real computer
Input
ML/I written in machine language of V1
Input
Real computer's macro assembler
Output
ML/I in the machine language of the real computer
Supplied by the author of ML/I
Supplied by the real computer's manufacturer
Used as the macroprocessor of Fig. 7-9

Fig. 7-10. Getting a general purpose macro processor running on a new computer.

When someone decides to write a portable program using this method, he must also decide what instructions his virtual machine is to have. A virtual machine with a single accumulator register and one-address instructions to load the accumulator, store the accumulator, add a memory word to the accumulator, and so on is easy to implement but inconvenient to write programs for. A machine with higher-level instructions (i.e., individual instructions that do more) is more convenient for programming the portable software, but it is harder to implement. Also, the lower the instruction level, the worse the ultimate code produced, as a general rule.

One way to handle this problem is for the writer of the portable software to design two virtual machines—a higher-level virtual machine and a lower-level virtual ma-

chine. His portable program, P, is written in the higher-level language and then translated to the lower level. A customer who wants P gets both versions.

The customer can implement the low-level version in a few days, due to its simplicity. Then he can begin working on a set of macros for the higher-level version. While he is trying to get these macros written, P is already available to use, although possibly inefficient. After some time, the super, fancy optimizing macros for the higher-level virtual machine will be finished, and the version of P produced from the low-level virtual machine can be put out to pasture. This two-level approach has the advantage of yielding a working version of P almost immediately, plus a better version later. While the better version is being developed, however, P can be used.

7.3.5. Emulation

When a computer center mothballs its old computer and gets a new one, the problem of portability becomes acute. Some of the users may have foreseen this transition and written all their programs for virtual machines that were carefully designed for portability. They will have few problems. Other users will have programmed only in problem-oriented languages. The latter will have some problems, as outlined in Section 7.3.1; but as long as their languages are available on the new machine, they will generally manage in some way.

For the programs written in the machine language of the old machine, or for compilers that translate special-purpose languages to the old machine language, the new machine spells disaster. These programs can be moved to the new machine provided that the old machine language is available as one of the new machine's levels. The most convenient way to accomplish this step is to provide the new microprogramming level with an interpreter for executing programs in the old machine language.

If this strategy is adopted, the new machine will have (at least) two microprograms: one for its own conventional machine level and one for the old machine's. The interpreter for the old machine is usually called an **emulator**, to distinguish it from the "native" conventional machine interpreter. On a computer with a general-purpose microprogramming level, such as the B1700, such a distinction is somewhat pointless, because there is no "native" conventional machine level. The microprogramming level has been specifically designed to be able to interpret a wide variety of virtual machines.

The most difficult part of an emulator's job is emulation of the old i/o instructions and devices. I/O instructions are normally carried out in parallel with CPU instructions and, unlike CPU instructions, i/o is strongly timing dependent. If the program being run in emulation made assumptions about relative speeds, ensuring that it works properly on the new computer can be difficult.

To illustrate the kind of problems that can arise, consider a program that sequentially reads records from a disk file into a buffer the same size as one record and processes them. To gain speed and save space, the program initiated reading the next record into the buffer before it had finished processing the old record, knowing that the new data would not actually begin overwriting the old data for X milliseconds, due

to the seek time of the old disk. It then used these X milliseconds to finish up its processing just in time. If the new disk seeks relatively faster (compared to the speed of the emulated CPU) than the old one, the old data may be erased before it has been processed. Mallach (1975) gives an excellent introduction to emulation techniques, particularly emphasizing i/o emulation.

7.3.6. Networks

Computer networks can also be used to solve the portability problem. The ideal network in this respect would have at least one of every important computer as a network node. If a user at a CDC installation finds a 370 assembly language program that he needs (in a book or journal, for example) he can simply punch it up and run it on one of the network's 370s. This solution is efficient with respect to machine time, requires no special preparation, and is not affected by timing problems. In the long run, a large-scale national or international computer network may provide a satisfactory solution to the portability problem.

7.4. SELF-VIRTUALIZING MACHINES

In a time-sharing or multiprogramming system, a single level 2 machine supports a number of level 3 machines by running each one in turn for a little while. Figure 7-11 depicts this situation. Machine 1 will be interpreted by the microprogram and

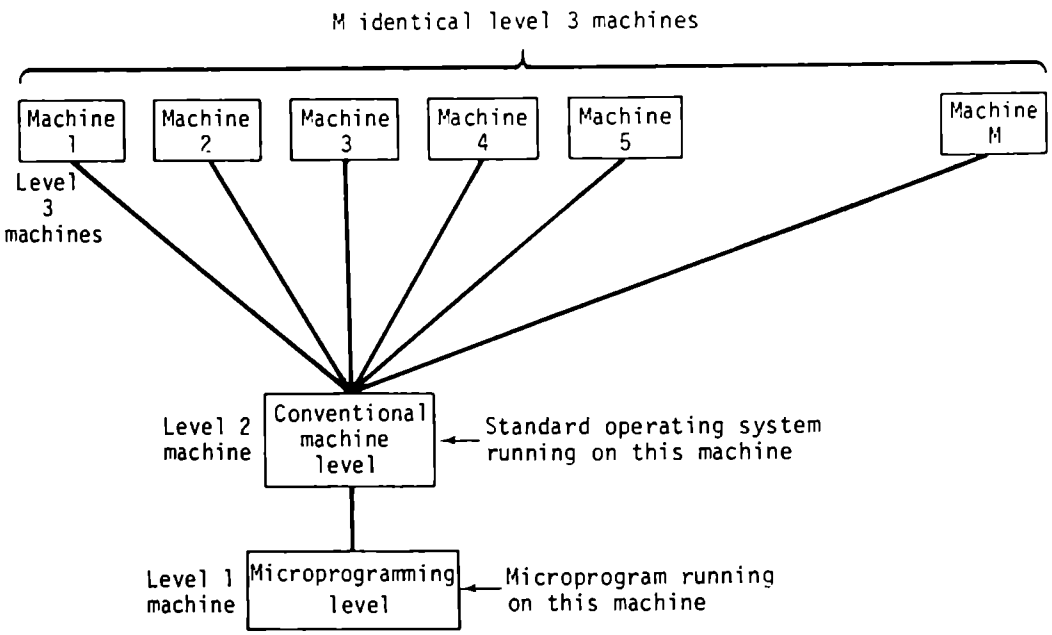


Fig. 7-11. A multiprogramming operating system supporting M level 3 machines in parallel.

operating system (depending on the instruction) for a short time interval, then machine 2 will be run, and so forth.

Usually the level 3 machines supported in this way have a major subset of the conventional machine level instructions (the "ordinary" ones but not the i/o instructions, HALT, etc.), as well as new virtual instructions for reading and writing files, managing directories, and communicating with other processes.

The instructions available at either level 2 or level 3 can be categorized as follows:

1. Available only at level 2 (e.g., command the disk to seek).
2. Available at both level 2 and level 3 (e.g., load register).
3. Available only at level 3 (e.g., read a record from a file).

Although the level 2 and level 3 machines have many instructions in common, the two machines are by no means identical.

The picture of similar, but not identical, level 2 and level 3 machines outlined above is not the only possibility, however. Another kind of operating system exists whose level 3 machines are not just similar to the level 2 machine but are exactly identical to it. On these level 3 machines, every single instruction available on the level 2 machine is also available on the level 3 machine, with no exceptions whatsoever. Each of the level 3 machines has the complete repertoire of instructions in categories 1 and 2 above. A program running on one of these level 3 machines has no way of telling that it is not running on the bare level 2 machine, except possibly by timing itself.

A multilevel machine in which two or more levels are identical is called a **self-virtualizing** machine. Such machines exist in practice as well as theory and have some interesting properties and applications, both of which will be briefly examined in the following sections.

7.4.1. IBM VM/370 System

An example of an operating system producing a self-virtualizing machine is the IBM VM/370 operating system, which is based on an earlier system for the 360/67, called CP-67. Figure 7-12 illustrates how VM/370 might be used.

The microprogramming level supports a single 370 level 2 machine (conventional machine level). This machine runs the VM/370 operating system, resulting in five (or any other number of) virtual 370s at the level above it. These virtual 370s are identical to the level 2 machine, even though they are running at level 3. To avoid confusion, we will call the conventional machine level 370s running at level 3 "virtual 370s." However, it should be kept in mind that the level 2 370 is also "virtual" in that it is interpreted by the microprogram. Each of the virtual 370s has the same instruction set as a "real" 370—that is, a level 2 370. The virtual 370's ordinary (category 2) instructions are interpreted directly by the microprogram. The i/o and other category 1 instructions are interpreted by VM/370. An operating system that supports virtual copies of the machine it itself runs on is called a **virtual machine monitor.** VM/370 is an example of a virtual machine monitor.

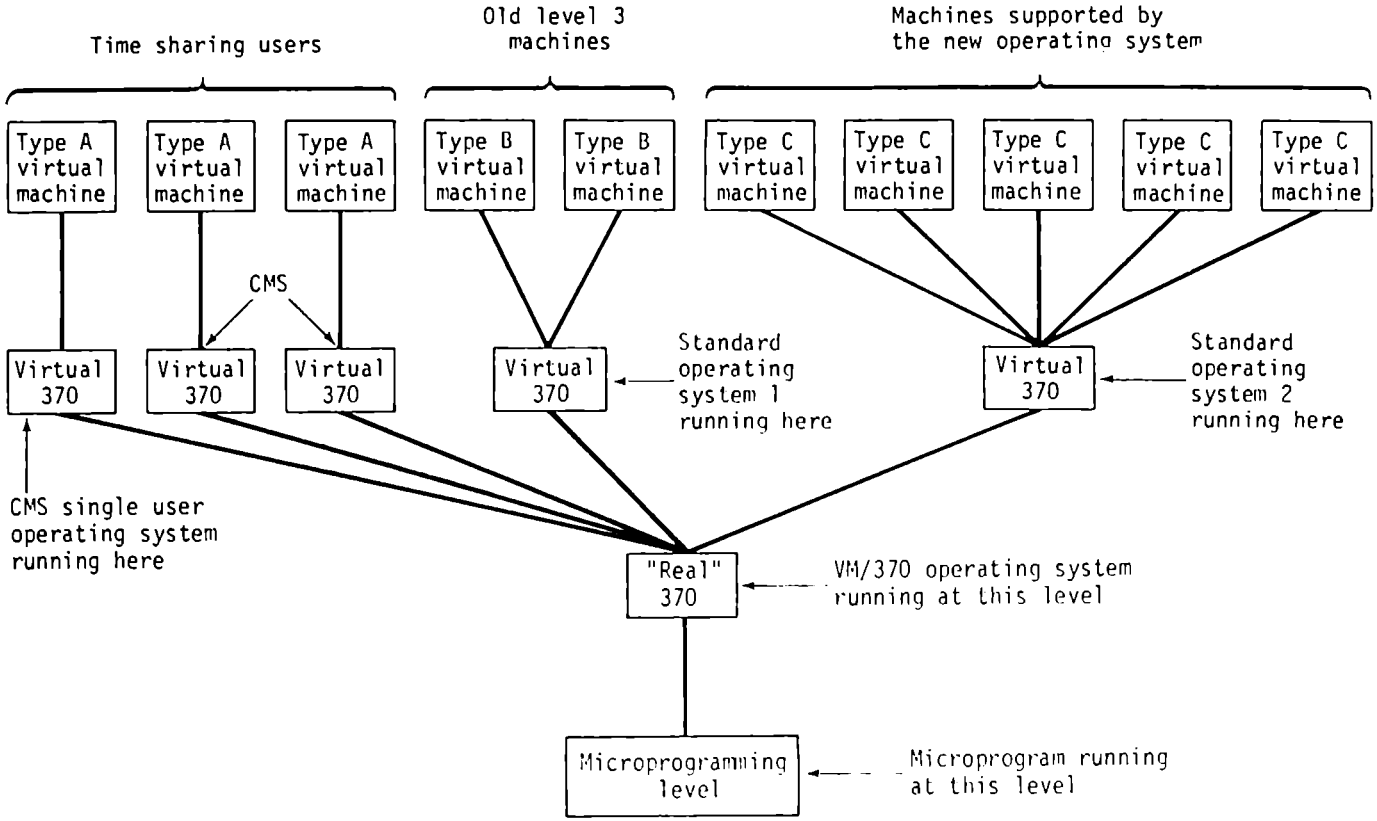


Fig. 7-12. VM/370 supporting five virtual machines in parallel, all of them identical to the 370 conventional machine level.

The virtual 370s are run in parallel, just as the level 3 machines of Fig. 7-11. A simple scheduling algorithm might run the first machine for 100 msec, then the second for 100 msec, then the third, fourth, and fifth, each for 100 msec, and then the first again, and so on indefinitely. It is important to remember that each of the virtual 370s supported by VM/370 is a normal 370 just as described by the "IBM System/370 Principles of Operation" manual, with the exception of its timing. Peculiarities in the timing are due partly to the multiprogramming and partly to the special handling given to the category 1 instructions, as described later.

7.4.2. Goals of Self-Virtualizing Machines

Since each of the virtual machines supported by VM/370 can run any 370 conventional machine level program, there is no reason why it cannot run any of the standard 370 operating systems. Nor is there any reason why different virtual 370s cannot run different operating systems. Figure 7-12 shows how three different operating systems can be used simultaneously.

IBM has produced a number of standard operating systems that support multiprogramming (and, in some cases, time-sharing). (These systems include DOS (for the smaller models), OS (for the larger models), VS1 (a simple operating system with virtual memory), and VS2 (a sophisticated multiprogramming system with virtual memory), among others. The original intention was probably that they all be compatible with one another, but it has not quite worked out that way. Furthermore, each one has passed through many (sometimes incompatible) versions.

As a consequence of the plethora of operating systems, the following situation often happens. A computer center has been using a certain operating system for awhile, and all its users have programs that run on the (level 3) virtual machines it provides. A new operating system whose level 3 machines offer some attractive features becomes available. The users would like to use the new operating system but are not enthusiastic about converting all their old level 3 programs.

What is needed is a multiprogramming system that provides some virtual machines identical to the old level 3 machines and some virtual machines identical to the new level 3 machines. These machines are called type B and type C in Fig. 7-12. The standard operating systems referred to are multiprogramming systems, such as OS and VS2, which support virtual machines with file i/o and all the other OSML instructions discussed in Chapter 5. The machines at level 4 in Fig. 7-12 are usually found at level 3 instead.

It is interesting to note the similarity between emulation and the use of self-virtualizing machines to support two operating systems in parallel. The purpose of emulation is to allow two different level 2 machines to be run in parallel by multiprogramming a single processor. VM/370 allows running two or more different level 3 machines in parallel, by creating multiple virtual 370s and running different operating systems on them. In both cases, two or more different machines can coexist. In one case, they are level 2 machines, and in the other they are level 3 machines.

Self-Virtualizing Machines and Time Sharing

One of the major goals of having self-virtualizing machines is to allow simultaneous operation of several different operating systems, but it is not the only one. A time-sharing system based on a standard operating system is a very complicated beast because it must perform two different functions:

1. Handle the multiprogramming.
2. Interpret the OSML virtual i/o and other instructions.

The multiprogramming management deals with activities like deciding which process (virtual machine) is to be run next, preparing it to be run (e.g., loading its registers into the hardware registers), running it, saving it, and deciding what to run next. Interrupt handling is also part of the multiprogramming. Interpreting the OSML instructions is unrelated to the multiprogramming activities.

The VM/370 operating system allows these functions to be completely separated. VM/370 can be used to produce a large number of bare virtual 370s. Each of these virtual 370s can run a single-user operating system that need only interpret OSML instructions and need not worry about multiprogramming. IBM's CMS (Cambridge Monitor System or Conversational Monitor System) is a single-user system that is used in conjunction with VM/370. The virtual machine supported by CMS is labeled type A in Fig. 7-12.

A complete time-sharing system can be constructed using VM/370 to support many bare virtual 370s, each of which runs CMS. By dividing a standard operating system's functions into two parts—the VM/370 part to handle multiprogramming and the CMS part to handle OSML interpretation—each of the two parts becomes smaller and simpler. Considering the enormous complexity of a standard operating system like VS2, any simplification is enormously useful.

Operating System Testing

A third goal of self-virtualizing machines follows from the ability to run multiple operating systems simultaneously. One or more of the virtual 370s can be used to test a new version of an operating system while other virtual 370s are running production jobs and providing time-sharing service, using CMS. Should the new version crash to a halt, there will be no effect on the production jobs or time-sharing service. What will happen is that VM/370 will continue to run each virtual 370 in turn, for a certain length of time.

Whether a particular virtual 370 actually does any useful work during its time quantum, or simply sits there halted, is of no concern to VM/370. When the time quantum is up, VM/370 moves on the next virtual 370. A virtual 370 coming to a halt under VM/370 is analogous to a user program coming to a halt under an ordi-

nary time-sharing system. In neither case are any of the other virtual machines running in parallel affected.

The virtual 370s supported by VM/370 can run any 370 operating system, including VM/370 itself. This ability can be exploited for testing new versions of VM/370 without disturbing production jobs. Figure 7-13 shows VM/370 running itself. This situation is sometimes called a recursive virtual machine.

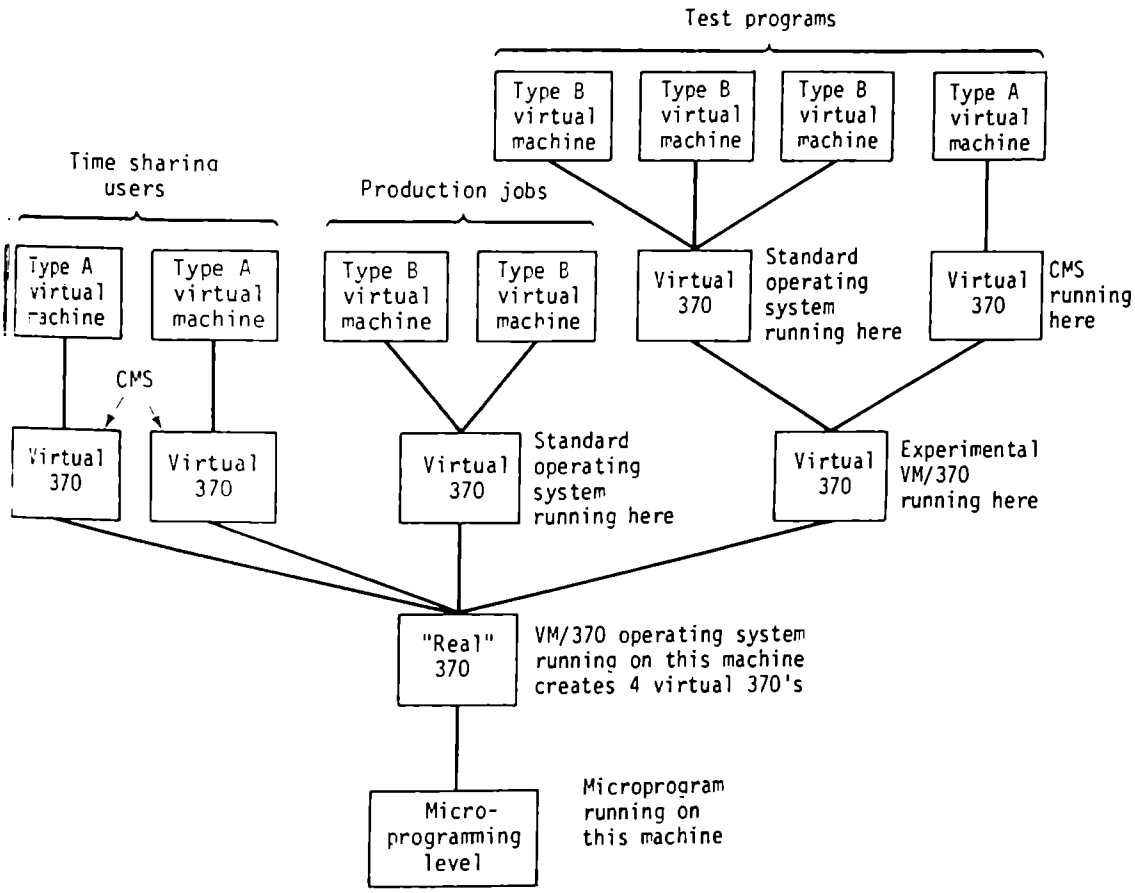


Fig. 7-13. VM/370 running VM/370 in one of the virtual 370's.

Protection of Confidential Data

Because each of the virtual 370s is functionally equivalent to a real 370, there is no way for a process running on one virtual 370 to even know about the existence of, let alone be able to spy on, another virtual 370. Consequently, VM/370 can provide a highly secure environment for handling confidential data, such as that possessed by banks, hospitals, law enforcement agencies, credit bureaus, and other institutions that maintain personal data about people. Standard operating systems are usually much less secure.

7.4.3. Implementation of a Self-Virtualizing Machine

A typical third-generation computer, such as a 370 or a PDP-11 (but not a Cyber 70), has two (or three) modes of operation at the conventional machine level. On the 370, such modes are called supervisor state and problem state. On the PDP-11, they are called kernel mode and user mode. Sometimes they are called master and slave mode. In supervisor state, all instructions are allowed. In problem state, only those instructions regarded as category 2 in Section 7.4 are allowed (i.e., the ordinary instructions). Supervisor state was intended for running the operating system at level 2. Problem state was intended for making sure that level 3 user programs were prevented from executing i/o and other instructions that might interfere with other users.

The instructions that are prohibited in problem state are called **privileged instructions**—the category 1 instructions of Section 7.4. They include

1. I/O instructions.
2. Instructions that affect the virtual address mapping.
3. Instructions affecting memory protection.
4. Instructions that load, store, or test the program status word.
5. Instructions that read or set clocks or timers.

The PDP-11 has no instructions to perform such activities, but it has memory addresses whose contents affect them. Thus the PDP-11 has privileged addresses instead of privileged instructions, but the effect is similar.

All operating systems, including VM/370, run their level 3 programs in problem state and protect any privileged addresses that the machine may have. If a level 3 program (in problem state) tries to execute a privileged instruction, the microprogram refuses to interpret it and instead starts up the operating system so that it can handle the situation. On a standard operating system, a message like

```
ATTEMPT TO EXECUTE PRIVILEGED INSTRUCTION AT LOCATION 3B6D
```

is printed and the job is aborted.

Exceptions and Virtual Machine Faults

For a virtual 370 being supported by VM/370, this situation is inadequate. A virtual 370 always runs with the real (level 2) machine in problem state, in order to trap any attempts to actually perform i/o. However, logically, the virtual 370 may be in either problem state or supervisor state, just as a real 370 may be in either problem state or supervisor state. If a virtual 370 executes a privileged instruction when it is logically in problem state, it must act exactly like a real 370—that is, cause a trap to the operating system in its machine. When it executes a privileged instruction while logically in supervisor state, VM/370 must simulate the privileged instruction.

These two situations are illustrated in Fig. 7-14 for the case of VM/370 supporting two virtual 370s, both running CMS. When the user program is running, the virtual

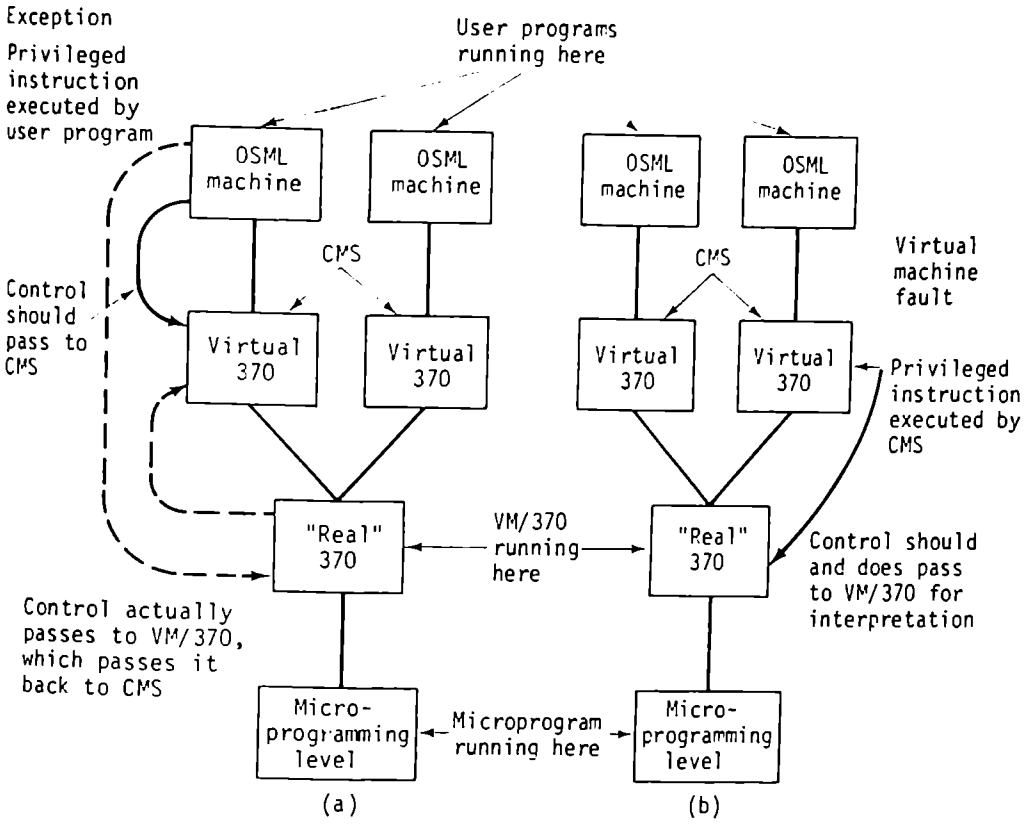


Fig. 7-14. (a) An exception. (b) A virtual machine fault.

370 is logically in problem state, because a real 370 is in problem state while running user programs. When the virtual 370 is running the operating system (in this case, CMS), it is logically in supervisor state, because a real 370 is in supervisor state while running the operating system.

At the time the privileged instruction actually occurs, the real hardware 370 will be in problem state (in order to catch all privileged instructions). The microprogram does what it always does when a privileged instruction is executed in problem state. It stores the PSW in location 40 and loads a new PSW from location 104. This starts VM/370 running.

VM/370 must then look at its tables to determine which virtual 370 was running and which state it was in. In other words, it must update its tables whenever a virtual 370 switches states, in order to keep an accurate record of which virtual 370s are logically in problem state and which are logically in supervisor state. If the virtual 370 was logically in problem state, VM/370 must reflect the trap back to the virtual machine and start CMS running with the virtual 370's PSW taken from the virtual machine's location 104. Goldberg (1973) calls this situation an **exception.** It is illustrated in Fig. 7-14(a).

If, however, the virtual 370 was logically in supervisor state at the time of the

trap, as shown in Fig. 7-14(b), then VM/370 must simulate the instruction. After all, the operating system in the virtual 370 will need to perform i/o occasionally, just like any other operating system. Goldberg (1973) calls this situation a **virtual machine fault**, to distinguish it from an exception. Note that their causes and treatments are very different. Ideally, the microprogram should give control to the operating system in the virtual machine when an exception occurs, and not to VM/370. In fact, an optional modification to the microprogram for handling virtual machine faults and exceptions more efficiently is available.

Simulation of Virtual Machine I/O

Most of the privileged instructions can be carried out in a straightforward way. Virtual i/o devices are mapped onto physical i/o devices because VM/370 may support N virtual 370s, each of which thinks it has a complete disk. Of course, if each actually fills up its entire disk, VM/370 really needs N disks to handle the data. Normally a virtual 370 uses only a small portion of its virtual disk, in which case VM/370 can allocate the available real disk space among the virtual disks. When virtual machine 4 writes onto cylinder 25 of its virtual disk, VM/370 can keep this data wherever it wants, provided that it can find the data again when virtual machine 4 tries to read cylinder 25 back in again.

This mapping of virtual disk addresses is similar to the operation of a segmented virtual memory, with the virtual disk number being the segment and the cylinder being the page. VM/370 must keep tables specifying the mapping. Similarly, virtual printers can be mapped onto real printers or, if there are insufficient real printers, onto disk files for subsequent printing.

Self-Modifying Channel Programs

One aspect of the 370's i/o method causes considerable difficulty for VM/370. On a real 370, the CPU normally prepares an i/o channel program and then executes a START I/O instruction to cause one of the channels to begin executing it. When a virtual 370 executes a START I/O instruction, a virtual machine fault occurs and VM/370 must carry out the i/o.

The i/o channels on the 370 use real addresses, not virtual addresses. As a rule, this fact poses no problems. VM/370 examines the i/o channel program instruction by instruction, replacing the virtual disk addresses by the actual disk addresses and the virtual machine's memory addresses by the corresponding real addresses. Then VM/370 starts one of the real i/o channels executing the modified i/o channel program.

One minor problem can arise if a single channel instruction reads in a block of data that crosses a page boundary. Since consecutive virtual pages need not be adjacent in real memory, the data may have to be read into two or more widely separated areas of real memory. Unfortunately, the i/o channel hardware cannot switch buffers in the middle of a channel instruction. VM/370 must check for this possibility and break up any instructions that cross page boundaries into several i/o instructions, each of which is for one contiguous block.

The real trouble comes from the fact that an i/o transfer may overwrite part of the channel program, thereby changing the channel program before it is finished. An even bigger problem is that the IBM operating systems make heavy use of this feature for performing virtual i/o on files. VM/370 has no way of properly modifying those parts of a channel program that are read in by the execution of the channel program itself. It can only fix up those parts of the channel program that are in memory at the time that the START I/O instruction is executed. With certain restrictions (self-modifying channel programs must be arranged so that the virtual disk addresses are the same as the actual disk addresses), VM/370 struggles through.

Shadow Page Tables

Self-modifying channel programs are not the only problem caused by peculiarities of the 370 organization. Because real 370s have virtual memory, virtual 370s must also have virtual memory. To illustrate the difficulties that can arise when supporting a virtual 370 with virtual memory, consider the use of VM/370 to run VS2 on a virtual 370. A change has been made to the page-replacement algorithm used by VS2, and a test run is being made to see if it works. Keep in mind the fact that VS2 thinks it is running on a real 370.

Just before VS2 gives control to one of the user programs in its machine, it sets up the user program's page tables to map the user program's virtual addresses onto what it considers to be real memory. The box in Fig. 7-15(a) shows a four-page user program mapped onto the virtual 370's memory. The virtual 370's memory is mapped onto a portion of the real 370's memory as shown.

The hardware of a 370, real or virtual, has an internal register that points to the segment table for the current program. The segment table points to the page tables. For simplicity, only the page tables are shown in Fig. 7-15. When an operating system is about to run a user program, it must set this register to point to the segment table of that program.

When VM/370 needs to run a user program on one of the virtual 370s, which page tables should it set this register to? Not the page tables in the virtual 370, since they map the user's program onto the virtual machine's memory, not onto real memory. [These two are the same only if VM/370 keeps the virtual 370's memory in one contiguous block starting at real address 0, which is not the case in Fig. 7-15(a).]

To solve this problem, VM/370 maintains a **shadow page table** that maps the user program's virtual addresses onto the real memory addresses where the pages are located. This mapping is shown in Fig. 7-15(b). VM/370 constructs the shadow page table from the page table in the virtual 370, plus its knowledge of where it is keeping the virtual 370's memory in real memory.

Because VS2 thinks it is running on a real 370, it carries out all the memory management functions associated with paging. These functions include reading and writing pages, as well as making the corresponding changes to the page tables in its memory. Changes to the page tables are accomplished by using ordinary load and store register instructions, which do not cause virtual machine faults. Consequently, any change

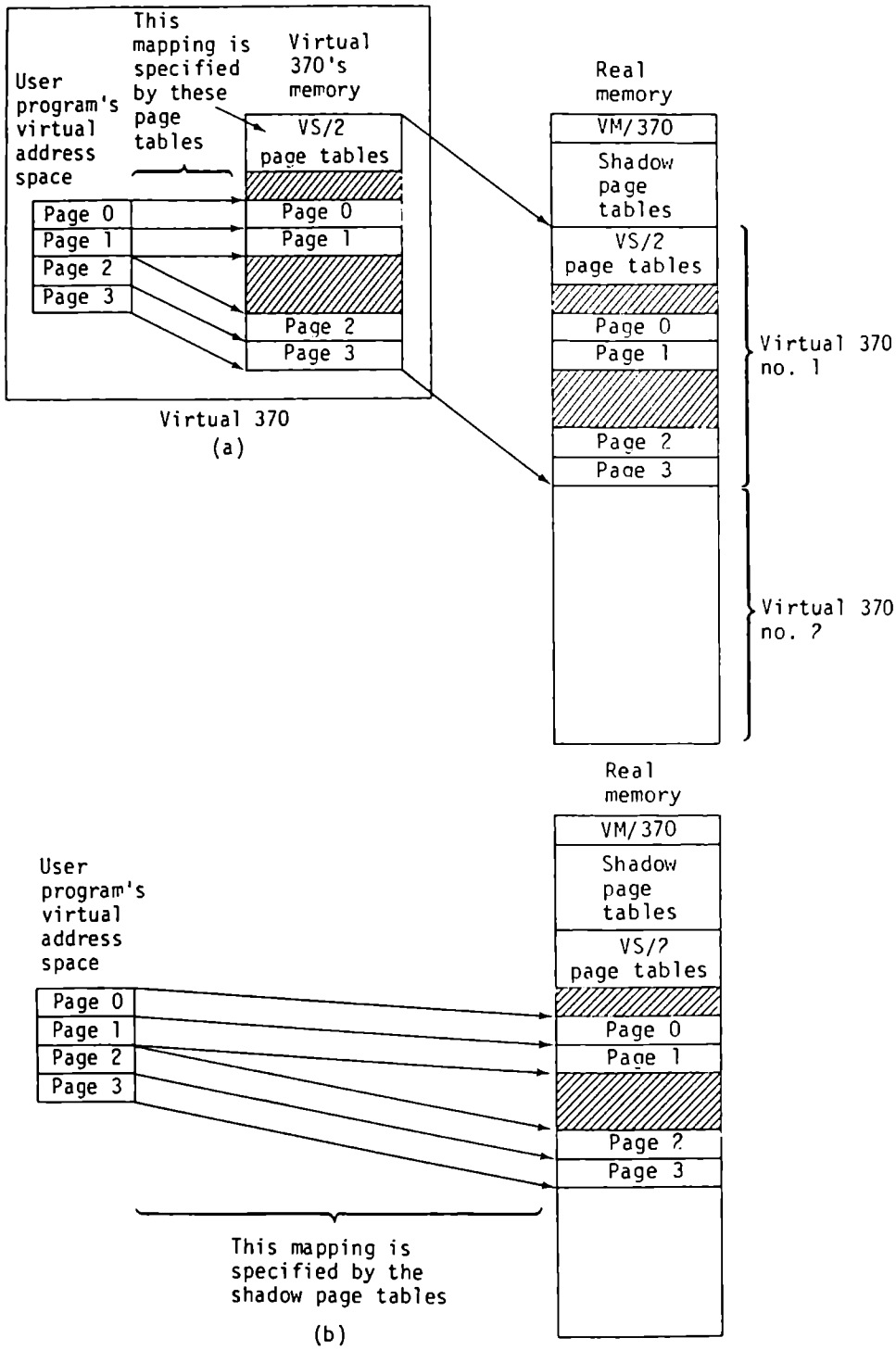


Fig. 7-15. The shadow page tables.

to a page table invalidates at least one of VM/370's shadow page tables, without VM/370 being aware that it has happened. VS2 can hardly be expected to inform VM/370 that it has changed its page tables, because it thinks it is running on a real 370 and may, in fact, do so later.

Fortunately, there is a solution to this problem. Like MULTICS, the 370 has a high-speed associative memory (see Fig. 5-42) that keeps track of the locations of the most heavily used pages. Whenever the page tables are changed, the operating system must erase the associative memory so as to force the hardware to reload it dynamically from the modified page tables. The instruction to erase the associative memory is privileged and causes a virtual machine fault. When the virtual machine fault occurs, VM/370 constructs new shadow page tables. As long as VS2 is properly programmed and always erases the associative memory after modifying the page tables, everything will work fine. If VS2 fails to erase the associative memory, it will not work on either a virtual 370 or a real one.

7.5. HIGH-LEVEL MACHINE ARCHITECTURE

When a programmer wants to run a program written in PL/I or some other problem-oriented language on a typical multilevel computer, the following steps occur. First, the program is read and analyzed by the compiler. Then the compiler generates an object program to be run on the level 3 machine. Finally, the object program is interpreted by the microprogram and the operating system, the former handling the ordinary instructions and the latter handling the file i/o and other OSML instructions.

In effect, the level 3 machine defines an interface between the compiler and the interpreters. This situation is illustrated in Fig. 7-16. The compiler translates source programs to this interface level, and then these programs are interpreted. The architecture and instructions of this interface level are not determined by the hardware but by the compiler and interpreters. If a new interface level were defined, the compilers would have to be modified to produce object programs for it, and a new interpreter (microprogram) would be needed to interpret it; but, in principle, the old CPU could still be used.

In a sense, the design of the compiler-interpreter interface level is arbitrary. Of course, a particular CPU may be most efficient at interpreting a certain class of virtual machines, so the designer is not completely free if efficiency is the dominant consideration, which it often is. Nevertheless, the microprogramming level of many computers is flexible enough to be able to support virtual machines that differ appreciably from the conventional machine level described throughout this book.

Once one has realized that the design of the compiler-interpreter interface level is not uniquely determined by the hardware, certain questions naturally arise:

1. What is the best interface level?
2. Should different programming languages use different interface levels?
3. If the level 2 and 3 machines currently in use are not optimal, why are they used?

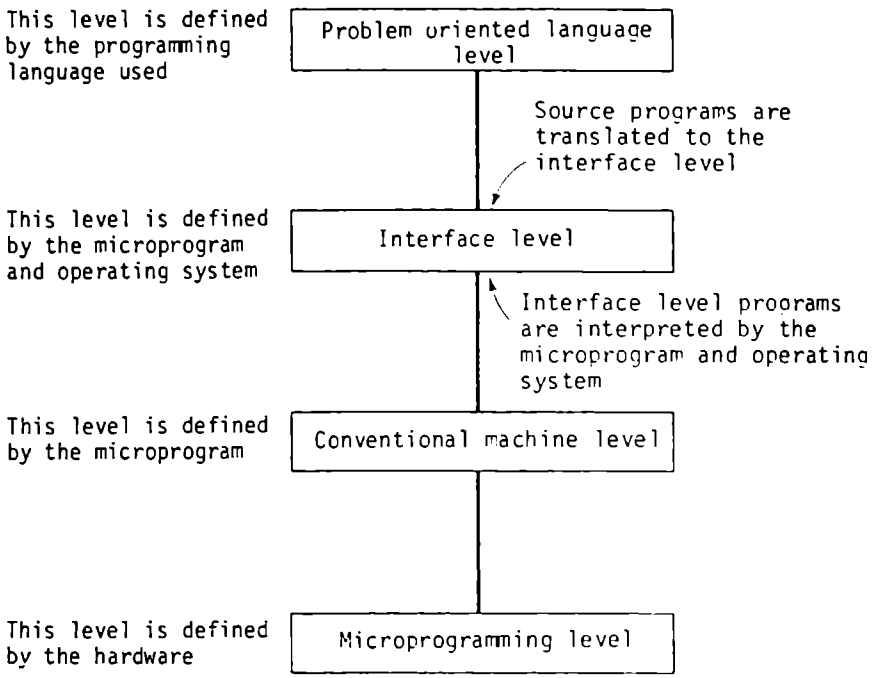

This level is defined by the programming language used
Problem oriented language level
Source programs are translated to the interface level
This level is defined by the microprogram and operating system
Interface level
Interface level programs are interpreted by the microprogram and operating system
This level is defined by the microprogram
Conventional machine level
This level is defined by the hardware
Microprogramming level

Fig. 7-16. The interface between the compiler and interpreter.

At the moment no one really knows what kind of interface is optimal. This subject is an area of active research, and the reader interested in the state-of-the-art should consult the current literature. Still, certain trends are clear. The biggest break with traditional machine design is the growing use of interface levels that are much closer to problem-oriented languages than the conventional machine level designs ever were.

These interface levels are often called **high-level machines**. The approach used on the B1700, for example, is to provide a different interface for each programming language in use, in order to tailor the machine to the programming language rather than the other way around. Burroughs calls these high-level machines "S-machines" (S = Software or Source or System). Figure 7-17 shows how this concept of a high-level machine relates to multilevel machines.

For simplicity, in this figure and in the following sections, the operating system is to be ignored, since the essential point here is the direct interpretation of the high-level instructions by the microprogram and not the i/o, virtual memory and other features traditionally supported by the operating system. It is also to be expected that in the future many of the features currently supported by the operating system will be handled directly by the microprogram interpreting the high-level machine.

A programming language consists of two basic parts: syntax and semantics. The syntax is the set of rules that determine which combinations of symbols are allowed and which are not. The PL/I syntax allows

```
DO WHILE (K < N); K = K + K; END;
```

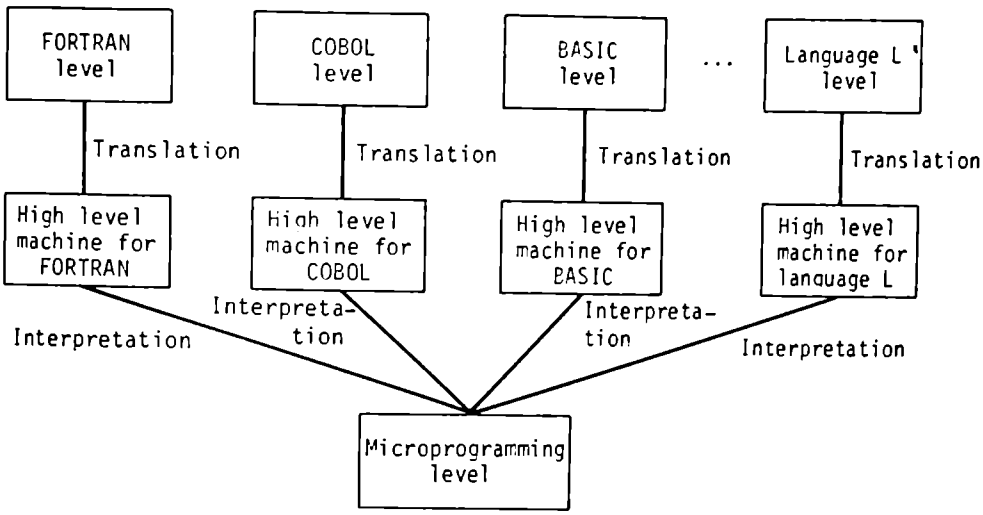


Fig. 7-17. The use of several microprograms on the same hardware CPU, each one supporting a different high level machine.

but prohibits

```
WHILE K < N DO K := K + K OD;
```

whereas the ALGOL 68 syntax allows the latter and prohibits the former, even though they have the same meaning.

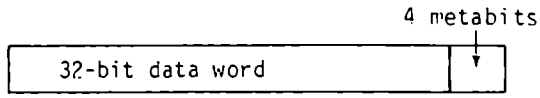
The semantics, on the other hand, specify what the various statements mean—in other words, what actions are to be performed. The preceding two statements have different syntax but perform the same actions. In languages like PL/I, the basic actions include assignment, conditional execution (IF statements), calling procedures, and so forth. A typical high-level machine instruction has a trivial syntax (a binary opcode, possibly followed by a few address fields), but the action performed is very similar to some action found in problem-oriented programming languages.

In the following sections we will briefly examine some ideas relevant to high-level machines.

7.5.1. Addressing and Descriptors

The use of metabits (see Section 2.2.3) to distinguish different kinds of data has several advantages over not having the type stored along with the data. Figure 7-18 shows a 32-bit word with four metabits. These metabits could be used to specify the contents of the word, as indicated.

A single ADD instruction could be used to add binary or decimal integers, complex numbers, floating-point numbers, and possibly even entire arrays. A three-address ADD instruction would give the addresses of the two operands and the result. The interpreter would be responsible for determining the types of the operands and performing the correct type of addition. This process might involve conversions, between binary and decimal, between integer and floating point, or between floating



Metabits	Meaning
0000	Binary integer
0001	Decimal integer
0010	Binary floating point number
0011	Decimal floating point number
0100	Complex number
0101	Character
0110	Boolean
0111	Pointer
1000	Dope vector for a binary integer array
1001	Dope vector for a decimal integer array
1010	Dope vector for a binary floating point array
1011	Dope vector for a decimal floating point array
1100	Dope vector for a complex array
1101	Dope vector for a character array
1110	Dope vector for a boolean array
1111	Unspecified

Fig. 7-18. Use of metabits to distinguish different data types.

point and complex, for example. Transferring the responsibility for type recognition and conversion from the compiler to the interpreter can greatly simplify the compiler (at the price of increasing the interpreter complexity somewhat).

The use of metabits for type identification can be generalized to provide the basis for a powerful addressing mechanism. Associated with each variable and constant is a word called a **descriptor**, consisting of the type bits and the address of the variable of constant. A sample PL/I program is shown below.

```
DESC:  PROCEDURE OPTIONS(MAIN);
    DECLARE (I,J,M(1:1000),N(1:1000)) FIXED BINARY,
            (R,S,T(1:1000)) FLOAT BINARY,
            OMEGA COMPLEX;
    .
    .
    .
    R = I + S;
    .
    .
    .
    T = M - N; /* ARRAY ARITHMETIC */
    .
    .
    .
END DESC;
```

The descriptors for this program are shown in Fig. 7-19.

In this example integer arrays are assigned different types than integers or floating-

Descriptor number	Type	Address
0	Binary integer	Address of I
1	Binary integer	Address of J
2	Binary integer array	Address of the dope vector for M
3	Binary integer array	Address of the dope vector for N
4	Binary floating point	Address of R
5	Binary floating point	Address of S
6	Binary floating point array	Address of the dope vector for T
7	Complex	Address of OMEGA

Fig. 7-19. Descriptors for the eight variables in the program.

point arrays. The address part for an array points to the dope vector for the array. Because the type bits are contained in the descriptor, no metabits are needed in the data itself. On some high-level machines, the descriptors contain other information, such as the size of the object pointed to, or a pointer to the symbolic name of the variable, for use in printing symbolic dumps for debugging purposes.

During the execution of the program, memory is divided up into three areas: the program, the descriptors, and the data area. Because they are conceptually different, they may have different word lengths. (The microprogram can simulate any word length or lengths needed.) The output from the compiler is used to load all three areas. It is important to note that the descriptors are actually present in memory during the execution of the program and can be examined by the microprogram. The size of the data area determines the number of bits in the address part of a descriptor.

Since there is a one-to-one correspondence between descriptors and variables, any variable can be addressed by giving the index of its descriptor. In Fig. 7-19 a three-bit number, 0 to 7, is sufficient to address any variable. Consider, for example, the PL/I statement

```
R = I + S;
```

On a three-address, high-level machine with a six-bit opcode, each address requires only three bits to indicate the necessary descriptor, because the example program has only eight variables. The total instruction length needed to perform the foregoing PL/I statement is 15 bits (assuming that the interpreter handles the type conversions). The complete high-level machine instruction is shown in Fig. 7-20. Note the resemblance to Cyber 70 format 1 in Fig. 3-19.

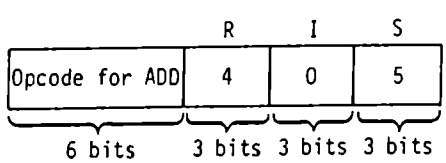


Fig. 7-20. A 3 address ADD instruction to compute R = I + S.

For comparison purposes, the Cyber 70 conventional machine level instructions needed to carry out the preceding PL/I statement are shown below.

```
SA3   I          30 bits   load I into X3
SA5   S          30 bits   load S into X5
PX3   X3         15 bits   convert I to floating point
NX3   X3         15 bits   finish converting I
FX6   X3 + X5    15 bits   X6 = I + S
NX6   X6         15 bits   normalize result
SA6   R          30 bits   store I + S in R
                 150 bits
```

The Cyber 70 requires 10 times as many bits to express the same statement. Not only is considerable space wasted compared to the descriptor approach, but it takes more time to fetch and decode seven instructions than one instruction. Furthermore, a compiler faced with the statement R = I + S; could produce Fig. 7-20 from it much more easily than it could produce the equivalent conventional machine level program shown above. The simplification in the compiler may be even greater than it appears at first, for the choice of registers X3 and X5 above may have resulted from a long register optimization calculation.

As a second example, consider the array assignment statement

```
T = M - N;
```

A three-address high-level machine instruction to do this is shown in Fig. 7-21. The equivalent Cyber 70 conventional machine level program to perform this is given below.

```
      SB1   M - 1      30 bits   B1 = address of M - 1
      SB2   N - 1      30 bits   B2 = address of N - 1
      SB3   T - 1      30 bits   B3 = address of T - 1
      SB4   1000       30 bits   B4 = 1000
L     SA2   B1 + B4    15 bits   X2 = M(B4)
      SA4   B2 + B4    15 bits   X4 = N(B4)
      IX6   X2 - X4    15 bits   X6 = M(B4) - N(B4)
      PX6   X6         15 bits   convert X6 to floating point
      NX6   X6         15 bits   finish conversion
      SA6   B3 + B4    15 bits   T(B4) = M(B4) - N(B4)
      SB4   B4 - 1     30 bits   B4 = B4 - 1
      GT    B4,B0,L    30 bits   IF B4 > 0 THEN GOTO L
                       270 bits
```

The high-level machine version requires 3001 main memory references (1 for the instruction and 3000 for the data), whereas the Cyber 70 version requires 6002 main

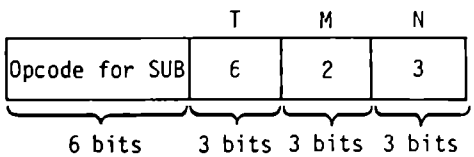


Fig. 7-21. A 3 address SUB instruction for the array computation T = M − N.

memory references (the loop is contained in three 60-bit words). Moreover, in the high-level machine version, only one instruction need be decoded; in the Cyber 70 version, 8004 instructions must be decoded. In addition, there is a factor of 18 difference in memory space required for the two programs. The main point here is simply that high-level machines may not only permit a simplification of the compiler but may yield run time gains as well.

The descriptor approach has disadvantages as well as advantages. The descriptors themselves take up space, and their fetching takes time. If a descriptor contains 4 type bits and 17 address bits (for comparison with the Cyber 70, which has a maximum of 2^{17} words of memory), then 21 bits are wasted for each descriptor. A program with 100 variables needs 2100 bits worth of descriptors, and an address field within an instruction requires 7 bits to be able to address all 100 variables (with 28 unused combinations). The instruction of Fig. 7-21 would need 27 bits instead of 15, using the more realistic assumption of 100 variables instead of 8 variables. The high-level machine using descriptors now requires 27 bits to compute $R = I + S$ or $T = M - N$, compared to 150 and 270 bits for the Cyber 70 versions. Assuming an average 183-bit saving for each PL/I statement, a one-page program (60 statements) would save 10,980 bits over the Cyber 70 version, and longer programs would save even more. This advantage more than offsets the memory needed for the descriptors.

The time lost fetching descriptors can be reduced by keeping the descriptors in the CPU's high-speed local storage. In addition, fetching three descriptors for the instruction of Fig. 7-20 takes less time than fetching the additional six instructions of the Cyber 70 version, even if the descriptors are kept in main memory. The interpreter also wastes some time doing type checking at run time, but basically this operation is a very simple one. Moreover, a high-level machine interpreter is larger and more complicated than an interpreter for a conventional machine level.

In order to optimize the descriptor addressing, the compiler must count the number of variables (and constants) used by the program, and from this number determine the number of bits in an address field. If there are 2^N variables and constants, an N-bit address field is needed. The compiler can then compile instructions using the proper size addresses. The address size must be passed to the interpreter so that it can decode the instructions properly. If this method is employed, the instruction lengths used by different programs will, in general, be different. The ability of the microprogramming level to handle address fields whose length is different for different programs becomes crucial here. (For contrast, note that a 370 conventional machine level program that does not need any displacements larger than 63 must still use a 12-bit displacement field in RX instructions.)

7.5.2. High-Level Machine Instructions

A variety of methods are used to encode programs for direct interpretation. One method is to store an image of the source program, with each token (variable, constant, =, WHILE, DO, ; etc.) represented by an integer. This system puts the full burden of the syntax analysis on the interpreter, but it requires a very trivial compiler.

A second method is to convert the entire program to reverse Polish and evaluate it using a stack. All the actions of the source language are turned into operators. For example,

```
IF DOG < CAT THEN BITE = 0; ELSE BITE = 1;
```

has the following reverse Polish representation:

```
    PUSH DOG      push DOG onto the stack
    PUSH CAT      push CAT onto the stack
    LESS          pop CAT and DOG and replace by 0 (false) or 1 (true)
    IF L1         pop top of the stack; go to L1 if it is 0
    PUSH BITE     push address of BITE onto the stack
    PUSH 0        push 0 onto the stack
    ASSIGN        pop address of BITE and 0 and set BITE = 0
    JUMP L2       statement is finished
L1: PUSH BITE     push address of BITE onto the stack
    PUSH 1        push 1 onto the stack
    ASSIGN        pop address of BITE and 1 and set BITE = 1
L2:
```

Note that all the operands (DOG, CAT, BITE, 0, BITE, 1) have been pushed onto the stack in their order of appearance, a general property of reverse Polish notation.

There are many variations on this theme—for example, pushing the descriptors for the operands onto the stack instead of the operands themselves. The basic idea is to rearrange the program to get it into the order needed for execution, without changing the meaning of the operators or statements.

The third method is to produce a true intermediate language to be interpreted. The three-address instruction of Fig. 7-21 is an example of one instruction that might be included in the language of this machine.

The IF instruction for this machine might have three addresses—the address of a variable to test to see if it is 0 or 1 and two other addresses, one for the THEN part and one for the ELSE part. Such an instruction is shown in Fig. 7-22. If the variable

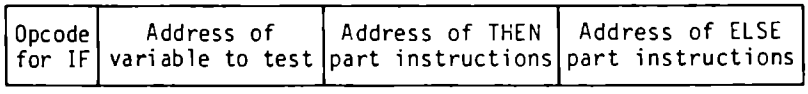


Fig. 7-22. An IF instruction.

has the value 1, the procedure at the first address is called; otherwise the procedure at the second address is called. Since there are no parameters to be passed, this procedure call requires only two steps:

1. Save the program counter on a stack.
2. Move the selected address into the program counter.

The THEN and ELSE procedures finish by popping the return address into the program counter.

A statement like

```
IF K < 0 THEN ... ELSE ...
```

might generate a three-address LESS instruction to compare K to 0 and put 0 or 1 in a temporary variable, followed by a three-address IF instruction to test the temporary variable and call the THEN or ELSE part procedure.

Procedure calls in the source language could be handled by pushing the descriptors of the parameters onto a stack and then pushing the return address onto the same or a different stack and transferring to the procedure. The machine design and compiler could be organized so that a procedure with N parameters could use descriptor numbers from 0 to N − 1 to access the actual parameters. Local variables would begin with descriptor N.

By adding extra bits to these descriptor addresses, variables declared in surrounding blocks could also be accessed. For example, in a block-structured language, such as PL/I, a given procedure may be nested within three surrounding procedures (including the main program). Two extra bits in the high-level-machine's address fields would suffice to select one of the four sets of descriptors, thereby allowing the innermost procedure to access variables in any of the surrounding ones.

The instruction repertoire available depends strongly on the programming language for which the machine is being used. If array operations are an important part of the language, as in APL, then the machine should have instructions for adding and subtracting vectors and matrices, multiplying matrices, and so on. The gains here simply in terms of the reduction in fetching and decoding instructions can be substantial.

Not all the high-level machine's instructions will be executed equally frequently. Space can therefore be saved by having several different opcode lengths, with the first two or three bits of the opcode indicating the length. The most frequent instructions should be assigned the shortest opcodes, and the least frequent instructions should be assigned the longest opcodes. For reducing the total number of opcode bits to the absolute minimum, a Huffman code (see Section 2.3.5) can be used. The B1700 S-machines use variable-length opcodes to minimize the number of bits required.

7.5.3. Advantages and Disadvantages of High-Level Machines

One of the major advantages of using a high-level machine is the resulting simplification in the compiler. Since the target machine has essentially the same semantics as the source language, the compiler has considerably less work to do. Consequently, it can be smaller, faster, easier to write, easier to understand, and more reliable.

At many computer centers, more than half the computer time is wasted doing compilations. When a user submits a program, he is interested in its execution, not in its translation to another language. Because the high-level machine matches the programming language much better, much of this waste can be reduced.

The large gap between the semantics of the programming language and the instructions and idiosyncracies of present level 3 machines often causes traditional compilers to produce highly inefficient programs. The very existence of 8 or 16 general

registers is practically a mandate for the compiler to attempt to optimize their use, whereas there is almost certainly nothing in either the program or the programming language about 8 or 16 registers, much less their optimal usage. The result of this mismatch is that the compiler spends a great deal of time trying to solve problems created by the underlying machine architecture, problems that are irrelevant to the user's program.

The worst part is that compilers rarely succeed at bridging this gap, despite their time-consuming efforts. A good assembly language programmer can usually write code that outperforms that of even an optimizing compiler by a factor of 2 or 3. Abolishing the conventional machine level also abolishes the gap between it and the program.

It is an unfortunate, but all too common, occurrence that compilers contain errors years after they are pronounced finished. Greatly simplifying a compiler tends to reduce the number of errors it contains.

A compiler for a high-level machine is really more like an assembler than a true compiler. Its primary tasks are as follows:

1. Syntax analysis.
2. Object code generation (comparable to an assembler).
3. Converting symbolic names to binary addresses.
4. Converting constants to the internal representation.
5. Allocating storage for data.
6. Removing comments.

Simplifying the compiler is not the only advantage of a high-level machine. The execution time of a program may also be reduced if the interpreter can directly carry out those combinations of primitive actions required by the programming language. For example, in many programming languages, entire arrays can be assigned in a single statement of the form A = B. A single high-level machine instruction MOVE ARRAY can easily be made to outperform an explicitly programmed loop for the conventional machine level. The saving in instruction fetching and decoding alone can be substantial.

Another advantage of high-level machines over conventional machines is in programming debugging. When a program fails to behave as expected, the only help the programmer generally gets is a message like

```
ILLEGAL INSTRUCTION DETECTED AT ADDRESS 7A4F
```

and a hexadecimal dump of memory. Unless he understands all the details of how the compiler stores its programs and data, the dump is nearly useless. On a high-level machine, it is possible for the post mortem program to relate the error much more closely to the source program, a big help to the programmer.

Still other potential advantages of high-level machines are compiler portability and arithmetic of the required precision. If a compiler produces 370 assembly language code as output, it is of little use on machines other than a 370. If it produces

output for a high-level machine, the compiler can be used on any computer where this high-level machine has been implemented. Also, a high-level machine may offer the possibility of setting the number of bits in an integer to whatever the programmer needs, rather than forcing him to use 32-bit integers.

The principal disadvantage of a high-level machine is the greater complexity required in the interpreter. However, because interpreters are usually much simpler than compilers, the decrease in compiler complexity can be expected to more than offset the increase in interpreter complexity.

You may be wondering why the conventional machine level is still so widely used despite its many disadvantages. There are several reasons. For one, the subject of high-level machines is still a research topic, and any new idea takes a number of years of development to move from the research laboratories to the stage of being a commercial product.

Another reason is the inertia generated by years of doing things in a certain way. Back in the 1940s when the present-day conventional machine level was first designed, it was an excellent idea. It provided machines that were both possible to understand and feasible to build. When the correct operation of the computer depended on the simultaneous functioning of 20,000 unreliable, power-hungry vacuum tubes, designers were understandably hesitant to shift any complexity from software to hardware.

A powerful force working against change has been the desire to build machines that are compatible with earlier ones. If a computer design group is told that the new machine they are to produce must be functionally identical to one that the company has been selling for the past 10 years, it is extremely difficult for them to incorporate any of the new ideas about computer architecture that have been discovered since the original machine was designed. Consequently, compatibility is also a recipe for stagnation. However, the current state of hardware technology (such as inexpensive single-chip CPUs) and microprogramming technology, coupled with the widespread use of problem-oriented languages, has created a situation favorable to the development of high-level machines. The result, in turn, has generated commercial interest in microprogrammable CPUs specifically designed to support high-level machines.

PROBLEMS

1. Describe the following:
   a. top-down design
   b. UNCOL
   c. emulation
   d. self-virtualizing machine
   e. shadow page table
   f. descriptor
2. Imagine that you are to write a complex numerical analysis program on a computer with only a macro assembler and no floating-point instructions. Describe two different methods for accomplishing this task.
3. What are the advantages and disadvantages of the top-down versus bottom-up design?

4. A number of computer manufacturers use the UNCOL principle to provide compatability on fundamentally different computers. To ensure the compatability, they prevent customers from writing their own microprograms. What benefits does the customer get from this policy? What costs are associated with it?
5. CDC manufactures a number of nonmicroprogrammed computers (e.g., 3600, 6600, and Cyber 72), all of which have a FORTRAN compiler. Programs to solve numerical problems can be moved from one machine to another. Is this a solution to the portability problem using an UNCOL? If so, draw an UNCOL diagram for it.
6. How can the advantage of the UNCOL approach (portability) be combined with the advantages of using high-level machines (simple compilers, efficient execution)? (*Hint:* Consider the number of interpreters that would be needed by your solution.)
7. As a homework problem, Prof. Slavedriver assigns his students the task of writing a complete IBM 370 interpreter for the PDP-11 level 3 machine. This virtual 370 is then used to compile and execute a PDP-11 simulator written in PL/I. Draw a diagram showing all the levels of this system.
8. How does the do-it-yourself virtual machine approach to portability differ from having a single UNCOL?
9. Why do writers of portable software go to the trouble of writing hardware test programs for nonexistent computers?
10. Compare a time-sharing system based on a self-virtualizing machine with a "conventional" time-sharing system.
11. Consider the situation of an IBM 370 (level A) running VM/370, and thereby supporting several virtual 370s (level B), one of which is also running VM/370 to support a collection of even more virtual 370s (level C). The level C 370s run a simple single-user operating system, such as CMS. For each of the following situations, describe the sequence of steps that occurs.
    a. The user program at level C causes an overflow exception.
    b. CMS at level C executes a START I/O instruction.
12. The VM/370 system must deal with the possibility that one of its virtual 370s will initiate an i/o instruction that reads data into a buffer that spans several (virtual) pages. If these pages happen to be located in contiguous page frames in real main memory, the buffer address can still be changed and the instruction executed, using the real address of the buffer. If these pages are not contiguous, a more complicated solution must be used. Why does this problem not occur with a "normal" operating system, of the kind described in Chapter 5? (*Hint:* What kind of i/o instructions are available on each?)
13. In a block-structured language, such as PL/I, a program may not only access variables declared in its own block, but it may also access variables declared in surrounding blocks. Show how descriptor addressing can be extended to handle the case of four levels of blocks, each of which contains 16 variables. How many bits does an address require in this situation?
14. You have just sold your stamp collection and taken up high-level computer design as your new hobby. Since you enjoy challenges, you have decided to start with a PL/I machine. Design an instruction or instructions for DO loops with no WHILE part, such as DO I = 1 BY N TO K + 1;.

8 SUGGESTIONS FOR FURTHER READING AND BIBLIOGRAPHY

In the preceding seven chapters a large number of topics were discussed, in various degrees of detail. This chapter has been provided to assist those readers interested in pursuing their study of computer organization further. Section 8.1 contains a list of suggested readings arranged by subject. Section 8.2 is an alphabetical bibliography of all books and articles cited in this book.

8.1. SUGGESTIONS FOR FURTHER READING

8.1.1. Addressing and Instructions

Gear, *Computer Organization and Programming*, pp. 15–48 and 70–87

Covers memory, the CPU, 3, 2, 1, and 0-address instructions, stacks, instruction formats, indexing, and indirect addressing. The IBM 370 and PDP-11 are used as running examples to illustrate the addressing structure and kinds of instructions found at level 2. The informal style, together with the fact that the 370 and PDP-11 are used as examples, makes this a good reference.

Hellerman, *Digital Computer Systems Principles*, pp. 327–359

An entire chapter on addressing and instruction sequencing. The topics covered include the length of information units, instruction design criteria, addressing design criteria, immediate, direct, and indirect addressing, indexing, variable-length operands, stacks, tradeoffs between opcode bits and address bits, instruction sequencing, and i/o. APL is used to describe instructions and addressing.

Stone, *Introduction to Computer Organization and Data Structures*, pp. 47–83

An introduction to addressing and instructions, using the the IBM 370 and Hewlett Packard 2116 (a 16-bit minicomputer) as examples. Arithmetic, data movement, boolean, shifting, unconditional branch and conditional branch instructions are described informally in English and more formally in ALGOL 60. Addressing methods, including indirect addressing and indexing, are also covered. Numerous sample programs are given for both machines. It is worthwhile looking at this reference to compare the HP 2116 with the PDP-11, since both are 16-bit minicomputers and yet they are enormously different. The PDP-11 was a major breakthrough in minicomputer design, and the HP 2116 is a typical example of a traditional one-address machine with a rather crude addressing mechanism (by PDP-11 standards).

Wegner, *Programming Languages, Information Structures, and Machine Organization*, pp. 35–56

Instructions are categorized as information moving, transformational, logical, branching, or iterative, each of which is discussed in turn. Address computation, indexing, indirect addressing, and their applications are covered. Stacks and reverse Polish are given a brief but clear treatment.

8.1.2. Assemblers and Assembly Language Programming

Barron, *Assemblers and Loaders*

This 61-page minibook provides a self-contained easy-to-read introduction to assemblers and how they work. It covers assembly language features, symbol tables, one-pass assemblers, two-pass assemblers, and meta assemblers, as well as macros and loaders.

Corbató, "PL/I as a Tool for System Programming"

A convincing argument against assembly language programming based on several years experience with MULTICS at Project MAC. Corbató describes the reasons PL/I was chosen as the language in which to write MULTICS, plus the consequences of that decision.

Dardner and Heller, "Streamline Your Software Development"

This paper discusses measurements of program performance, showing that in many programs a very large fraction of the execution time is accounted for by a very small fraction of the program. This point strongly suggests writing programs in a higher-level language, testing them, and rewriting only the critical parts in assembly language, instead of writing everything in assembly language.

Donovan, *Systems Programming*, pp. 59–110

A highly detailed discussion of how an assembler works, with emphasis on algorithms and data bases used. Worth reading by anyone actually about to write an assembler. The text is copiously illustrated with flowcharts, tables, programs, and figures. Linear and binary searching, sorting, and hash coding are covered in considerable depth. The chapter is followed by 13 pages of well-thought-out problems.

Gear, *Computer Organization and Programming*, pp. 332–410

An introduction to the operation of two-pass assemblers. This chapter is less detailed than Donovan's. The topics include the fields of an assembly language statement, converting decimal numbers to binary, recognizing names, evaluating expressions, pseudoinstructions, base register assignment, entries, externals, and the assembly process itself. There is also a section on macros and how they are expanded. Chapter 10 is about searching and sorting.

Graham, "Use of High Level Languages for System Programming"

A discussion of the use of a high-level language (PL/I) for writing a large operating system. The conclusion is plain: even for systems programming, the use of a high-level language is preferable to assembly language.

Stone, *Introduction to Computer Organization and Data Structures*, pp. 85–113

A short introduction to assemblers, with emphasis on the Hewlett Packard 2116 assembly language and assembler.

Wegner, *Programming Languages, Information Structures, and Machine Organization*, pp. 107–144

This is a more theoretical treatment of assemblers than the other references, with emphasis on general principles and concepts rather than on specific assemblers. In addition to the usual subjects, several advanced topics not covered in the other references are discussed here. They include stack-structured symbol tables, table-driven programs, free and bound variables, destructive and pushdown definition, register optimization, resource allocation, and assembly time operations.

8.1.3. Binary Numbers and Arithmetic

Flores, *The Logic of Computer Arithmetic*

A detailed examination of the algorithms used by computers for performing binary arithmetic on integers and floating-point numbers. The emphasis is on high-performance algorithms. Four entire chapters are devoted exclusively to integer multiplication, and three entire chapters are devoted exclusively to integer division.

Garner, "Number Systems and Arithmetic"

A tutorial on advanced binary arithmetic concepts, including carry propagation, redundant number systems, residue number systems, and nonstandard multiplication and division. Strongly recommended for anyone who thinks he learned everything there is to know about arithmetic in sixth grade.

Knuth, *Seminumerical Algorithms*, pp. 162–290

A wealth of material about positional number systems, floating-point arithmetic, and multiple-precision arithmetic. All algorithms are specified carefully, both in English and in the assembly language of a hypothetical computer called MIX. This material requires and deserves careful study.

Reitwiesner, "Binary Arithmetic"

An imposing algebraic analysis of the performance of various algorithms used for 2's complement binary arithmetic. In the author's own words: "The text suffers from an exhaustive attention to details."

Stein and Munro, *Introduction to Machine Arithmetic*

A thorough treatment of computer arithmetic in a general radix, r. Although the emphasis is primarily on the theory of machine arithmetic, there are also a large number of carefully worked problems and examples illustrating the algorithms covered.

van Wijngaarden, "Numerical Analysis as an Independent Science"

An article about the differences between finite-precision arithmetic and the concepts of classical mathematics. The emphasis is on theorems of classical mathematics, such as the associative and distributive laws that do not hold in finite-precision arithmetic.

8.1.4. Character Codes, Redundant and Nonredundant

Hamming, "Error Detecting and Error Correcting Codes"

This is the original article on error-correcting codes.

Huffman, "A Method for the Construction of Minimum Redundancy Codes"

This is the original article on minimum redundancy codes. In addition to describing the algorithm, an informal proof is given that the code is optimal.

Martin, *Teleprocessing Network Organization*, pp. 62–95

A good introduction to error-correcting codes, including polynomial codes for correcting multiple errors. The text is amply illustrated with examples.

Peterson, *Error Correcting Codes*

Probably the most complete work on error-correcting codes to date. It is by no means easy reading, requiring a knowledge of Galois field theory in places. Among other topics, it covers linear codes, cyclic codes, Bose-Chaudhuri codes, and burst-correcting codes.

8.1.5. Computer Organization

Beizer, *The Architecture and Engineering of Digital Computer Complexes*

An advanced, but highly readable, two-volume treatise on computer organization. Volume 1 deals with addressing, instructions, memories, i/o devices, programming techniques, and assemblers, among many other topics. Volume 2 deals with design and analysis.

Bell et al., "Effect of Technology on Near Term Computer Structures"

This article shows how technology influences computer organization. Modularity,

microprogramming, cache memories, and highly parallel machines are among the topics covered.

Bell and Newell, *Computer Structures: Readings and Examples.*

A vast compendium of information covering just about every kind of computer organization devised by the mind of man. If the book had been typeset in textbook style, it would have exceeded 1300 pages. The authors use a notation called PMS to describe the interaction of processors, memories, switches, and other computer components, and another notation, called ISP, to describe instruction sets. Many of the 44 chapters are reprints of key papers in the development of computer organization. In addition to conventional machines, array processors, list processors, multiprocessors, stack machines, and desk calculators are covered. There is an excellent model-by-model analysis of the IBM 360 series, which culminates in a redesign of the series, based on multiprocessors. Over 400 entries are listed in the bibliography.

Foster, "The Next Three Generations"

A speculative and fascinating article about the future of computer organization. Foster attempts to extrapolate current trends to the year 2000. He predicts that the average computer will then directly interpret one or more problem-oriented languages, have a very wide word, have no visible registers, use decimal arithmetic, have a wired-in operating system, and be monoprogrammed rather than multiprogrammed. This is "must" reading for anyone interested in research about computer organization.

Nelson, "Microprocessor Architecture"

Tiny computers, each contained in a single integrated circuit chip are revolutionizing computer organization. Where special purpose hardware was previously needed, a complete miniature computer can now be used instead, programmed to perform the function previously done in hardware. This special issue of *Computer* reviews the state of the art of these single chip computers (microprocessors) as of mid 1974.

Thornton, "Design of a Computer. The Control Data 6600"

This unusual little book is a highly detailed description of how the CDC 6600 is organized and why it was so organized. Since the author was one of the 6600's designers, he is in an excellent position to write about it. There are chapters on the overall organization, circuits, memory, CPU, functional units, peripherals, and the (original) operating system. Some engineering background is required in order to understand this book fully.

8.1.6. The Conventional Machine Level

Abrams and Stein, *Computer Hardware and Software*

A somewhat superficial text on the conventional machine level and its programming. A large number of conventional machine level programs for a hypothetical computer are given.

Anagnostopulous et al., "Computer Architecture and Instruction Set Design"

This is an article about a research group in graphics that bought a general-purpose microprogrammable computer and set out to design and implement a good level 2 machine. Ultimately, they decided that the PDP-11 was the ideal architecture. After implementing the PDP-11 interpreter, they discovered that deficiencies in their level 1 machine's organization made the resulting PDP-11 too slow. The article is a case history of a "do-it-yourself" approach to the conventional machine level, and illustrates some of the potential pitfalls.

Blaauw et al., "The Structure of System/360"

A series of five detailed articles describing the IBM 360 written by the people responsible for designing it. The articles give some insight into why they made the decisions they made. No one can question the authors' knowledge of their subject. (See also Amdahl, 1964.)

Chu, *Computer Organization and Microprogramming*

A somewhat hardware-oriented book covering both the microprogramming level and the conventional machine level. It covers arithmetic in great detail, with chapters on fixed and floating point.

Gear, *Computer Organization and Programming*

A textbook for a second course in computer science. The coverage and level are similar to this book, which makes it a good reference. There are chapters on the conventional machine level, system software, principles of programming, input/output, interpretation and simulation, multiprogramming and multiprocessing, assemblers, searching and sorting, and compilers.

Hellerman, *Digital Computer System Principles*

An introduction to computer organization for advanced electrical engineering undergraduates.

Katzan, *Computer Organization and the System* 370

The emphasis here is on the IBM 370, not on computer organization. Entire chapters are devoted to the 370 models 135, 145, 155, and 165. This book might be used as an introduction to the manufacturer's manuals for people who really want to know things like what bit 6 of control register 14 on the 370/135 does. (It indicates that the time-of-day clock is broken.)

Stone, *Introduction to Computer Organization and Data Structures*

Although much of this textbook is devoted to data structures, the first four chapters are concerned with the conventional machine level. Many concepts are illustrated with assembly language programs for the Hewlett Packard 2116 minicomputer and the 370. The chapter on i/o is especially interesting.

Wegner, *Programming Languages, Information Structures, and Machine Organization*

A number of important ideas are discussed here, including representation of numbers, adders, registers, types of instructions, addressing, stacks, interrupts, i/o, and virtual memory. One minor shortcoming is the scarcity of figures, which makes reading it more difficult than it would otherwise be.

8.1.7. Deadlocks

Coffman et al. "System Deadlocks"

An excellent tutorial on the subject of deadlocks. The situations leading to deadlocks are characterized, and their prevention and detection are discussed.

Habermann, "Prevention of System Deadlocks"

An important original contribution to the literature about deadlocks. A formalism for describing deadlocks is developed, and three theorems are proven using it. These theorems form the basis for an algorithm telling when it is safe to allocate some requested resource and when it is not safe.

Holt, "Some Deadlock Properties of Computer Systems"

A discussion of a graph model for investigating the properties of deadlocks. The article is essentially a condensation of the author's Ph.D. thesis, and it requires some study. Six theorems based on the graph model are derived.

8.1.8. File Systems

Daley and Neumann, "A General Purpose File System for Secondary Storage"

The first published description of the MULTICS file system, written by its designers. It discusses the relation between files and segments, directories, links, and branches, and many other aspects of the file system.

Donovan, *Systems Programming*, pp. 407–432

A meaty introduction to file systems. It emphasizes file organization, data bases, and algorithms. Successively more sophisticated file systems are covered. Readers interested in learning more about virtual i/o using files will find this a good place to begin.

Gildersleeve, *Design of Sequential File Systems*

This 49-page minibook gives a nontechnical overview of various aspects of file system design. The emphasis is on business applications. File maintenance, record design, and blocking, among other topics, are discussed.

Judd, *Use of Files*

A short introduction to files and file systems. Topics covered include file structures, information representation, sorting, security, information retrieval, and documen-

tation. The interaction of files and programs is considered at some length. An extensive bibliography is included.

McDonell and Montgomery, "The Design of Indexed Sequential Files"

A technical article about how to implement an index sequential file system. It begins with a tutorial on ISAM. The problems of storage allocation on the disk are covered in detail, and sample calculations are given to show how to optimize the parameters of the design.

Roberts, "File Organization Techniques"

A tutorial on file organization techniques, including sequential, random, list-structured, and tree-structured files. The implementation of each type is discussed. An excellent reference. The bibliography has hundreds of entries.

Tsichritzis and Bernstein, *Operating Systems*, pp. 123–147

A good reference on virtual i/o using files, and how it is implemented. Allocation, access methods, i/o, and file system organization are covered. The authors show how a file system can be constructed in a structured way, as a hierarchy of levels.

8.1.9. High-Level Machines

Bashkow et al., "System Design of a FORTRAN Machine"

This article proposes a design for a high-level FORTRAN machine. The machine has instructions for assignment, IF, GOTO, DO, READ, PRINT, and a few others. Provision is made for handling subscripted variables. The memory organization, program loading, and interpreter are described in detail. The designers actually intended to build the machine in hardware instead of interpreting it in microcode, a reflection on the age of this article.

Chevance, "A COBOL Machine"

A brief introduction to a high level machine designed to run COBOL programs. The machine design, data descriptors and instructions are mentioned, but not described in much detail.

Hassitt et al., "Implementation of a High Level Language Machine"

This article describes an APL machine implemented by having an APL interpreter run on the IBM 360/25 microprogramming level. The authors describe the representation of programs, the representation of data, the statement analyzer, function calls, operators, subscripting, and i/o. There is enough information here to get a good idea of how it actually works. Some knowledge of APL would be helpful in reading it.

SIGPLAN, "Proceedings of a Symposium on High Level Language Computer Architecture"

A collection of 19 papers on high-level machines. There are several articles on the SYMBOL-2R computer, which directly executes a high-level typeless language.

Designs for machines that execute dialects of ALGOL 60 and PL/I are proposed. Various other aspects of high-level machine architecture, including metabits, descriptors, and parallel execution are presented. These papers are all original research contributions and require some study to understand.

Weber, "A Microprogrammed Implementation of Euler on IBM System/360 Model 30"

Euler is a problem-oriented language based on ALGOL 60. This article discusses the organization of the Euler machine, the interpreter, and the compiler that converts source programs to the intermediate language that is eventually interpreted. Some of the problems encountered are mentioned.

8.1.10. Input/Output

Gear, *Computer Organization and Programming*, pp. 218–277

An elementary introduction to input and output. Various i/o devices, such as paper tape, printers, and plotters, as well as secondary memory devices, such as disks, drums, and magnetic tapes, are covered. Interrupts and concurrent i/o are treated, and there is a good discussion of buffering.

Stone, *Introduction to Computer Organization and Data Structures*, pp. 165–201

A discussion of i/o programming in assembly language. Interrupts and coroutines are explained. The function and use of data channels are also covered. The chapter contains quite a few assembly language programs for the Hewlett Packard 2116 used as examples.

8.1.11. Linkers and Loaders

Barron, *Assemblers and Loaders*, pp. 38–47

A short but clear introduction to absolute loaders, relocating loaders, and linkage editors (linkers).

Donovan, *Systems Programming*, pp. 149–199

A very detailed, but nevertheless clearly written, chapter on linkers and loaders. Compile and go loaders, general loaders, absolute loaders, relocating loaders, direct linking loaders, and dynamic linking loaders are all mentioned. The design and operation of direct linking loaders form the heart of the chapter. The algorithms and data bases needed are described carefully and thoroughly. Read this before writing your next linking loader.

McCarthy et al., "The Linking Subprogram Language and Linking Loader"

One of the first articles about relocating loaders.

Presser and White, "Linkers and Loaders"

A tutorial on absolute loaders, relocating loaders, and linking loaders, with emphasis on the last. The article is highly readable, despite its use of IBM jargon.

8.1.12. Macros

Brown, "The ML/I Macro Processor"

This article discusses some highlights of a machine-independent general-purpose macro processor. The ML/I macro processor is intended as a preprocessor for other language processors, not necessarily assemblers. (See also Brown under 8.1.14)

Campbell-Kelley, *An Introduction to Macros*

A short survey of the uses of macros. Both assembly language macro processors and general-purpose macro processors are considered. There is a chapter on using macros for implementing portable software, and another chapter on the use of macros in high-level languages.

Donovan, *Systems Programming*, pp. 111–148

A detailed description of how to implement a macro processor. It would be of considerable interest to anyone planning to undertake such a task. Among other things, it deals with conditional macro expansion, nested macro calls, and macro definitions within macro bodies. Detailed flowcharts illustrate how a macro processor actually works.

Gear, *Computer Organization and Programming*, pp. 188–211 and 363–373

An introduction to the facilities provided by most macro assemblers and how they are implemented. There are sample macros for the 370 and PDP-11.

Kent, "Assembler Language Macroprogramming"

An introduction to the use and implementation of macros in the IBM 360 assembler, including conditional assembly, global variables, and attributes.

Strachey, "A General Purpose Macrogenerator"

A description of a general-purpose macro processor called GPM. Both the language and the macro processor are described in detail, with a number of diagrams.

Waite, "The Mobile Programming System: STAGE 2"

Another general-purpose macro processor. This one was intended as an aid to producing portable software, but it is also interesting as a macro processor. Macro calls are identified not by the macro name, as is the case with most macro processors, but by matching each statement against a list of templates. This results in a language in which the user may define his own syntax.

Wegner, *Programming Languages, Information Structures, and Machine Organization*, pp. 130–179

Section 2.6 deals with the implementation of macros in assemblers. Sections 3.1 to 3.4 deal with macros on a more theoretical level. There are also sections on the TRAC and GPM macro languages. The general flavor of this reference is very different from those on macro assemblers.

8.1.13. The Microprogramming Level

Broadbent, "Microprogramming and System Architecture"

An interesting survey of the applications of microprogramming, beginning with emulation and ending with the direct interpretation of high-level machines and microprogrammed operating systems. A number of microprogrammable computers are mentioned, and some current trends are noted.

Davies, "Readings in Microprogramming"

Two-thirds of this article is devoted to an introduction to the concepts of emulation and microprogramming. There are short discussions of microinstruction design and performance of microprogrammed computers. The annotated bibliography has 53 entries.

Dollhoff, "The Negative Aspects of Microprogramming"

This article points out some difficulties that microprogrammable computers can cause for users who are dependent on the software supplied by their computer's manufacturer. The principle trouble is that if the customer changes the microprogram, he is, in effect, redefining the conventional machine level, which will result in the manufacturer's software ceasing to function. The author proposes a solution which amounts to having the manufacturer design and support basic, advanced, and user tailored instruction sets.

Flores, "Lookahead Control in the IBM System 370 Model 165"

The microprogramming level on the 370/165 uses a form of pipelining to speed up instruction execution. This article describes the organization of the 370/165's microprogramming level, and shows how the pipeline works. The important problem of how to interpret conditional branch target instructions is covered in detail.

Flynn and MacLaren, "Microprogramming Revisited"

A convincing argument for using microprogramming as a method of implementing the conventional machine level. The authors cover microinstruction formats, architectural implications of microprogramming, and software design for microprogrammed computers. The emphasis is on concepts rather than details. Alternative computer organizations are considered.

Husson, *Microprogramming: Principles and Practices*

This 600-page treatise contains a wealth of information about microprogramming and the microprogramming level. The first third of the book is devoted to an introduction to the concepts, applications, and technology of microprogramming. The last two thirds consists of a highly detailed examination of four microprogrammed computers, the IBM 360/40, IBM 360/50, RCA Spectra 70/45, and Honeywell H4200. Readers lacking a background in electrical engineering should first read Rosin, Flynn and MacLaren, as well as Davies. An excellent bibliography on microprogramming containing hundreds of entries is included.

Rosin, "Contemporary Concepts of Microprogramming and Emulation"

An excellent tutorial on microprogramming. The reader wishing to continue his study of this subject is advised to begin here. The basic concepts are introduced by means of an example. This is followed by sections on the virtues of microprogramming, the properties of some existing microprogrammable computers, and current research topics and problems.

Rosin et al., "An Environment for Research in Microprogramming and Emulation"

A major part of this article is a description of an extremely general purpose microprogrammable computer, the QM-1. This machine has six distinct memories, programmable bus connections, and facilities for interpreting machines of any word length. The uses to which a general-purpose microprogrammable computer, such as the QM-1 (or the Burroughs B1700), can be put are discussed.

Tucker, "Microprogram Control for System/360"

An introduction to microprogramming, covering microinstruction format, microinstruction sequencing, and timing. The article is oriented toward the techniques used in the IBM 360 rather than abstract concepts.

Wilkes, "The Growth of Interest in Microprogramming"

A literature survey of articles about microprogramming, with an intention of tracing the historical development of the field. Since Wilkes invented the concept of microprogramming, he is in an excellent position to trace its development. A bibliography of 55 entries, spanning 19 years, is included.

8.1.14. Multilevel Computers

Brown, *Macro Processors and Techniques for Portable Software*

An introductory text on the subject of macros and their uses, particularly for implementing virtual machines. The book is divided into three parts. The first part covers a number of macro processors, including assemblers, PL/I's macro facility, GPM, TRAC, ML/I, and STAGE2. The second part is concerned with how macros can be used to implement new levels in order to make software portable. The last part is an in-depth study of an example of a piece of portable software.

Dennis, "The Design and Construction of Software Systems"

An interesting article about the concept of building a computer as a series of levels. Hierarchies, methods of constructing new levels, and function, correctness, and reliability of software systems are discussed.

Dijkstra, "The Structure of the "THE" Multiprogramming System"

A much discussed article about the construction of a multilevel virtual machine. The author claims that the software was rigorously proven to be correct. Such a claim has never been made for any other operating system and certainly represents a step forward in the art of constructing software.

Goos, "Hierarchies"

This article is about building virtual machines as a series of layers as a design strategy. The order of making decisions is considered. The relation between languages and levels is also examined.

Rosin et al., "A Research Environment for Microprogramming and Emulation"

Discussion of a machine using nanoprogramming as a basis for microprogramming and microprogramming as a basis for higher-level machines. An exposition of the functions of the various levels is presented. The potential advantages of this kind of organization are examined.

Tsichritzis and Bernstein, *Operating Systems*, pp. 169–191

A chapter on design methods for multilevel machines. Both the top-down and bottom-up approaches are discussed.

Wilner, "Design of the Burroughs B1700"

An introduction to the Burroughs B1700, an advanced machine intended as the basis of multilevel hierarchical machines.

Zurcher and Randell, "Iterative Multilevel Modeling—A Methodology for Computer System Design"

An original research contribution introducing the top-down design method.

8.1.15. Networks

Farber, "Networks: An Introduction"

The subject of networks is introduced by means of examples. Seven existing networks are examined, ranging from the huge ARPA network, with dozens of nodes, to local networks, entirely restricted to one physical site. The goals, topology, and unusual features of each network are described.

Heart et al., "The Interface Message Processor for the ARPA Computer Network"

A good introduction to the role and implementation of the IMPs in the ARPA network, including both hardware and software aspects. There are a number of figures and even a photograph of an IMP.

Pyke and Blanc, "Computer Networking Technology—A State of the Art Review"

This nontechnical review deals primarily with communication channels and network architecture. Effects of the composition, size, topology, channel type, and control of a network are also considered.

Roberts and Wessler, "Computer Network Development to Achieve Resource Sharing"

An excellent introduction to networks in general and the ARPA network in particular. Network properties, such as reliability, responsiveness, capacity, and

topology, are described in general and then for the ARPA network. The ARPA network is compared with alternate designs.

Rustin, *Computer Networks*

This volume consists of the proceedings of a symposium on computer networks. There are 11 papers, most of which are descriptive rather than theoretical. Five networks are discussed in detail: IBM Computer Network/440, MERIT, OCTOPUS, CYBERNET, and ARPA. The papers describe the philosophies, organization, and implementation of these networks. The informal flavor of the original talks has not been lost, which makes this book a good introduction to the subject of networks.

Sanders, "Networking, an Overview"

As the title suggests, this brief article gives an introduction to networks. The main emphasis is on the successes and failures of existing networks, and predictions for the future. This article is followed by three nontechnical articles dealing with various aspects of networking, including experimentation, interuniversity collaboration, economics of networking, the future of networking, airlines reservations, and the use of a university network by outside customers.

Wessler and Hovey, "Public Packet-Switched Networks"

This nontechnical article is concerned with the communications aspects of computer networks, and the benefits that computer networks are likely to yield to customers and vendors alike.

8.1.16. Operating Systems

Barron, "Job Control Languages and Job Control Programs"

The author compares contemporary job control languages, primarily IBM's and CDC's to "the grunts by which Neanderthal Man communicated with his fellows." He describes what is wrong with them (the job control languages, that is, not the Neanderthal Men), and pleads for better ones in the next generation of machines.

Barron and Jackson, "The Evolution of Job Control Languages"

An introduction to job control languages, with emphasis on the ones used by the IBM 360 and ICL 1900. The former is shown to be like an assembly language, whereas the latter is more like a problem-oriented language.

Coffman and Denning, *Operating Systems Theory*

A highly mathematical textbook on operating systems. The emphasis is on analytical modeling of operating systems, to enable their performance to be predicted and understood.

Corbató and Vyssotsky, "Introduction and Overview of the MULTICS System"

A general introduction to the design goals of MULTICS. This article introducing MULTICS to the world was written by two of its chief designers long before the system was completed. One of their intentions was to put their ideas into print before

carrying them out, in order to see later on how far the system drifted from its original plans.

Hansen, *Operating System Principles*

An introductory textbook on operating systems, with an emphasis on processes and parallel processing. There are also chapters on processor management, memory management, scheduling algorithms, and protection, as well as a very detailed case study of the RC4000 operating system. The treatment is more mathematical than the book by Madnick and Donovan or the book by Tsichritzis and Bernstein, but it is less mathematical than the book by Denning and Coffman.

Madnick and Donovan, *Operating Systems*

A good introductory textbook on operating systems. The emphasis is more on the resource management functions of an operating system than on the virtual machine aspects. The emphasis is on multiprogramming systems, such as those used on the 370.

Organick, *The MULTICS system: An Examination of Its Structure*

A clearly written, but highly technical, description of the inner workings of MULTICS. The book contains many detailed descriptions of tables, algorithms, and procedures used by MULTICS. There are chapters on the virtual memory, dynamic linking, the calling sequence, protection, the file system, interprocess communication, and i/o. A comprehensive MULTICS bibliography is included.

Shaw, *The Logical Design of Operating Systems*

An introductory text on operating systems, emphasizing process management, resource management, and file systems. The treatment is general and is not tied to any specific system. System organization, batch systems, processors, multiprogramming, memory management, sharing, resource control, deadlocks, and file systems are each the subject of separate chapters. Many algorithms are illustrated by short programs in ALGOL 60.

Tsichritzis and Bernstein, *Operating Systems*

Another good introductory textbook on operating systems. The first part of the book emphasizes the main concepts, including processes, processor allocation, memory management, virtual memory, file systems, and protection. The second part of the book covers design techniques and other practical information on actually constructing an operating system.

8.1.17. Parallel Processing

Bell et al., "Effect of Technology on Near Term Computer Structures"

A tantalizing survey of current research in computer organization, emphasizing highly parallel machines. Topics covered include the Carnegie Mellon multimini, the ARPA network, the ILLIAC IV, pipeline processors, and associative processors.

Dijkstra, "Solution of a Problem in Concurrent Programming"

A first attempt at synchronization of parallel processes in ALGOL 60. The solution is rather cumbersome, but it does demonstrate that a solution is possible without any special primitives.

Dijkstra, "Cooperating Sequential Processes"

A widely cited paper introducing the mutual exclusion, or synchronization, problem and its solution using semaphores. A generalized mutual exclusion problem is introduced and solved. Numerous examples fill this informative and entertaining article.

Dijkstra, "Hierarchical Ordering of Sequential Processes"

An extensive discussion of synchronization of parallel processes using semaphores. The problem of the dining philosophers leads to the relationship between the directors and the secretaries.

Foster, *Computer Architecture*, pp. 169–220

These three chapters deal with the problem of making the computer run faster by means of parallelism. Distributed, associative, and cellular computers are covered. Examples are drawn from the CDC 6600, IBM 360/85, IBM Stretch, Honeywell 8200, and Edinburgh design.

Hansen, "Concurrent Programming Concepts"

This tutorial on parallel programming first carefully develops the concept of a sequential process. Then it shows the difference between structured and unrestricted concurrency. The remainder of the article is devoted to synchronization of parallel processes. The emphasis is on the control structures needed in programming languages to express parallelism. Numerous examples are given.

Lorin, *Parallelism in Hardware and Software*

An informal introductory textbook on parallel processing. It covers multiple machines, multiprogramming, multiprocessing, i/o, operating systems, and processes, among other things. A good introduction for the reader who wants to learn a lot about parallel processing, both in hardware and software.

Wirth, "On Multiprogramming, Machine Coding, and Computer Organization"

The semaphore operations are proposed as an alternative to having interrupts. Several detailed examples of the use of semaphores are given, and their implementation on the IBM 360 is discussed. A clearly written article.

8.1.18. Self-virtualizing Machines

Buzen and Gagliardi, "The Evolution of Virtual Machine Architecture"

An excellent introduction to the concept of a self-virtualizing machine (which they call a virtual machine) and how it has evolved. The authors cover virtual machines and emulators, process state mapping, i/o mapping, memory mapping, dual and single state architectures, virtual machine faults, and dynamic map composition.

Buzen and Goldberg, "Virtual Machine Techniques for Introducing Peripherals into Computer Systems"

One application of a self-virtualizing machine is to ease the introduction of a new i/o device into a computer system. As an example of this technique, they cite the introduction of the IBM 2314 disks at the virtual machine monitor level, while the operating systems in the virtual machines continued to manipulate the old 2311s. Several other examples are given.

Goldberg, "Architecture of Virtual Machines"

Third-generation computer systems are all, to a greater or lesser degree, unsuited for supporting virtual copies of themselves. Their microprograms make no distinction between exceptions and virtual machine faults, and there are problems keeping the shadow page tables accurate, among other difficulties. This article proposes a machine architecture that solves these and other problems in hardware (or in the microprogram).

Goldberg, "Workshop on Virtual Computer Systems"

A collection of research papers on self-virtualizing computer systems.

Goldberg, "Survey of Virtual Machine Research"

An excellent introduction to the history, principles and current practices concerning self-virtualizing machines. The problems posed by third generation architectures and some solutions to them are discussed. A section on performance covers sources of overhead and various ways of reducing it, including impure machines. The paper concludes with comments about the uses of self-virtualizing machines in education, and for enhancing software reliability and data security.

Parmelee et al., "Virtual Storage and Machine Concepts"

The first part of this article is about virtual memory and the second part is about CP-67. It covers i/o simulation, channel program translation, and the CP-67 virtual memory. An annotated bibliography of 96 entries is included.

Popek and Goldberg, "Formal Requirements for Virtualizable Third Architectures"

A model of self-virtualizable computer systems is developed, showing what conditions must be present for a third-generation system to be able to support virtualizable machines. Intuitively, the requirement is that any instruction that must be interpreted by the virtual machine monitor must cause a trap when executed in problem state. On some computers, these sensitive instructions are simply ignored in problem state.

8.1.19. Symbol Tables

Knuth, *Sorting and Searching*

The definitive work on the subject of sorting and searching. Anything not said here is probably not worth saying. Particularly noteworthy is Knuth's analysis of algorithms, giving best and worst case analyses for all algorithms, as well as averages.

Lum et al., "Key to Address Transformation Techniques"

An empirical study of six different hashing techniques: division of key by prime number, digit analysis, midsquare, folding, Lin's method, and algebraic coding. Each technique was used to store and retrieve thousands of records from a number of large files in order to compare the techniques. The results are displayed in a series of tables. Their conclusion is that dividing the key by a prime number gives the best overall performance.

Martin, "Sorting"

An excellent tutorial on a variety of sorting techniques. Algorithms discussed in detail include bubble sort, Shell sort, quick sort, radix exchange sort, sample sort, address calculation sort, polyphase sort, and oscillating sort. The article includes a flowchart to aid the reader in choosing the proper sort for his application. The bibliography includes over 100 entries and references to 37 algorithms.

Morris, "Scatter Storage Techniques"

A good survey of hash-coding techniques applicable to symbol tables. Several methods for handling collisions are analyzed, and the expected performance for each is derived. The article also considers the implications of virtual memory on hash tables. Several sample programs for entering and retrieving data in hash tables are given.

Price, "Table Lookup Techniques"

A tutorial on searching techniques. Techniques discussed include the merge search, binary search, estimated entry search, and direct lookup. Key transformations are also covered.

Rich, *Internal Sorting Illustrated with PL/I Programs*

This book is an annotated collection of debugged PL/I sorting procedures. Programs are given for most of the usual techniques, as well as for more exotic ones, such as pigeon hole sort, ancestral sort, tournament sort, funnel sort, and tree sort. The results of actual timing tests for each procedure are given. Address sorting (a form of hash coding) is the fastest method, with a recursive sort (quicksort) a close second. It is interesting to note that a recursive procedure greatly outperforms a large collection of nonrecursive ones, contradicting the popular myth that recursion is somehow "inefficient."

Shell, "A High Speed Sorting Procedure"

Shell's original article describing Shell sort. The article is of interest not only for the algorithm, which is still popular, but for the 1959 style of exposition. It includes an assembly language procedure for the long-extinct IBM 704. The modern programmer not initiated into the mysteries of statements like HXJ U, 4,– – may enjoy the flowcharts in any case.

8.1.20. Telecommunications

Bacon and Bull, *Data Transmission*

A short but well written introduction to telecommunications. It covers basic communications concepts, character codes, modulation pulse techniques, error correction, parallel and serial transmission, store-and-forward networks, multiplexing, and interfaces, both synchronous and asynchronous. The book emphasizes the practical rather than the theoretical aspects.

Davies and Barber, *Communication Networks for Computers*

An introductory textbook on the subject of communications and data transmission. Both the public telephone network and private networks are considered. The topics covered include information encoding, multiplexing, packet switching, protocols, and reliability. Although communications software is explicitly studied, much of the text has a mild hardware flavor. The emphasis is on the concepts, however, and not on circuits or formal mathematics.

Martin, *Teleprocessing Network Organization*

A practical guide to telecommunications. The book covers many devices, coding techniques, and transmission methods. Not much theory, but many examples taken from existing systems. The author has also written several other books on the subject of telecommunications.

8.1.21. Virtual Memory

Belady, "A Study of Replacement Algorithms for a Virtual Storage Computer"

Replacement algorithms are divided into three groups—those depending on page-usage history, those depending on history of presence or absence of pages in memory, and those assuming that all pages are equally likely to be used at any time. Some characteristics of each group are given. An optimal algorithm is discussed.

Bensoussan et al., "The MULTICS Virtual Memory: Concepts and Design"

A clear introduction to the MULTICS virtual memory. It covers segmentation, paging, processes in MULTICS, segment attributes, segment addressing, faults, and multiplexing.

Chu and Opderbeck, "Performance of Replacement Algorithms with Different Page Sizes"

A study of several paging algorithms undertaken in order to determine which parameters most affect performance. Among other parameters, main memory size and secondary memory access time are considered. One conclusion is that for a secondary memory that is relatively fast (compared to main memory), small page sizes are better; if the secondary memory is relatively slow, large page sizes are better.

Daley and Dennis, "Virtual Memory, Processes, and Sharing in MULTICS"

An introduction to a number of the basic ideas in MULTICS, including address space, address formation, indirect addressing, intersegment addressing, and linking. The paper emphasizes these concepts rather than the specific details of their implementation.

Denning, "The Working Set Model for Program Behavior"

This is the original paper on the working-set model. In it, Denning first describes paging on computers that do not use this model of program behavior. Then the concept of the working set is carefully defined and analyzed mathematically. Scheduling, sharing, and resource allocation are also covered.

Denning, "Virtual Memory"

An excellent tutorial on virtual memory. Topics covered include segmentation, paging, storage utilization, replacement algorithms, page size, fragmentation, demand paging, and the working-set model. The bibliography contains 84 entries.

Fotheringham, "Dynamic Storage Allocation in the Atlas Computer, Including an Automatic Use of a Backing Store"

A brief description of paging on the Atlas computer, the first computer with virtual memory. It is worthwhile comparing this article with the one by Bensoussan et al. in order to see how much progress has been made since the idea of virtual memory was first conceived.

Lorin, *Parallelism in Hardware and Software*, pp. 435–499

Although the book is primarily about parallelism, this section deals with many of the important ideas of virtual memory. The author has a highly readable, almost chatty style.

Randell and Kuehner, "Dynamic Storage Allocation Systems"

An interesting treatment of virtual memory from a somewhat different point of view. There is a long section on storage addressing, in which the distinction between segmented and nonsegment memories is examined and comparisons between existing systems made. Storage allocation strategies are discussed. This article deals only with concepts and not with implementation details.

Saltzer, "A Simple Linear Model of Demand Paging Performance"

The author proposes a model of demand paging performance in which the time interval between successive page faults is linearly proportional to the amount of main memory available to a program. The model has been tested using MULTICS; these tests are discussed. Several applications of the model are described.

Wilkes, *Time Sharing Computer Systems*, pp. 46–75

A short introduction to paging, segmentation, associative memories, page-replacement algorithms, intersegment communication, and protection. The text suffers somewhat from a lack of figures.

8.2. ALPHABETICAL BIBLIOGRAPHY

Abbreviations: CACM—Communications of the ACM
FJCC—Fall Joint Computer Conference
SJCC—Spring Joint Computer Conference
NCC—National Computer Conference

ABRAMS, MARSHALL D., and STEIN, PHILIP G.: *Computer Hardware and Software—An Interdisciplinary Introduction.* Reading, Mass.: Addison-Wesley, 1973.

ACKERMAN, A. FRANK: "Quadratic Search for Hash Tables of Size p^n," *CACM*, vol. 17, pp. 164–165, March 1974.

AMDAHL, G. M., BLAAUW, G. A., and BROOKS, F. P., JR.: Architecture of the IBM System/360, *IBM Journal Res. Develop.*, vol. 8, pp. 87–101, April 1964.

ANAGNOSTOPOULOUS, P. C., MICHEL, M. J., SOCKUT, G. H., STABLER, G. M., and VAN DAM, A.: "Computer Architecture and Instruction Set Design," *Proc. NCC*, pp. 519–527, 1972.

BACON, M. D., and BULL, G. M.: *Data Transmission.* New York: American Elsevier, 1973.

BARRON, D. W.: *Assemblers and Loaders.* New York: American Elsevier, 1969.

BARRON, D. W.: "Job Control Languages and Job Control Programs," *Computer J.*, vol. 17, pp. 282–286, August 1974.

BARRON, D. W., and JACKSON, I. R.: "The Evolution of Job Control Languages," *Software Practice and Experience*, vol. 2, pp. 143–164, 1972.

BASHKOW, THEODORE R., SASSON, AZRA, and KRONFELD, ARNOLD: "System Design of a FORTRAN Machine," *IEEE Transactions on Computers*, vol. EC-16, pp. 485–499, August 1967.

BEIZER, BORIS: *The Architecture and Engineering of Digital Computer Complexes.* New York: Plenum Press, 1971.

BELADY, L. A.: "A Study of Replacement Algorithms for a Virtual Storage Computer," *IBM Systems Journal*, vol. 5, pp. 78–101, 1966.

BELL, C. GORDON, and NEWELL, ALLEN: *Computer Structures: Readings and Examples.* New York: McGraw-Hill, 1971.

BELL, C. GORDON, CHEN, ROBERT, and REGE, SATISH: "Effect of Technology on Near Term Computer Structures," *Computer*, vol. 5, pp. 29–38, March–April 1972.

BELL, JAMES: "The Quadratic Quotient Method: A Method for Eliminating Secondary Clustering," *CACM*, vol. 13, pp. 107–109, February 1970.

BENSOUSSAN, CLENGEN, and DALEY, R. C.: "The MULTICS Virtual Memory: Concepts and Design," *CACM*, vol. 15, pp. 308–315, May 1972.

BLAAUW, G. A., BROOKS, F. P., STEVENS, W. Y., AMDAHL, G. M., and PADEGS, A.: "The Structure of System/360," *IBM Systems Journal*, vol. 3, no. 2, pp. 117–196, 1964.

BROADBENT, J. K.: "Microprogramming and System Architecture," *Computer Journal*, vol. 17, pp. 2–7, February 1974.

BROWN, P. J.: *Macro Processors and Techniques for Portable Software.* New York: Wiley, 1974.

BROWN, P. J.: "The ML/I Macro Processor," *CACM*, vol. 10, pp. 618–623, October 1967.

BUZEN, J. P., and GAGLIARDI, U. O.: "The Evolution of Virtual Machine Architecure," *Proc. NCC*, pp. 291–299, 1973.

BUZEN, JEFFREY P., and GOLDBERG, ROBERT P.: "Virtual Machine Techniques for Introducing Peripherals into Computer Systems," *Proceedings COMPCOM 74*, IEEE Computer Society, 1974.

CADOW, H.: *OS/360 Job Control Language*. Englewood Cliffs, N.J.: Prentice-Hall, 1970.

CAMPBELL-KELLEY, M.: *An Introduction to Macros*. New York: American Elsevier, 1973.

CHEVANCE, R. J., "A COBOL Machine," *SIGPLAN Notices*, pp. 139–144, August 1974.

CHU, YAOHAN: *Computer Organization and Microprogramming*. Englewood Cliffs, N.J.: Prentice-Hall, 1972.

CHU, W. W., and OPDERBECK, H.: "Performance of Replacement Algorithms with Different Page Sizes," *Computer*, vol. 7, pp. 14–21, Nov. 1974.

COFFMAN, E. G., JR., and DENNING, PETER J.: *Operating Systems Theory*. Englewood Cliffs, N.J.: Prentice-Hall, 1973.

COFFMAN, E. G., JR., ELPHICK, M. J., and SHOSHANI, A.: "System Deadlocks," *Computing Surveys*, vol. 3, pp. 68–78, June 1971.

CORBATÓ, F. J.: "PL/I as a Tool for System Programming," *Datamation*, vol. 15, pp. 68–76, May 1969.

CORBATÓ, F. J., and VYSSOTSKY, V. A.: "Introduction and Overview of the MULTICS System," *Proc. FJCC*, pp. 185–196, 1965.

DALEY, ROBERT C., and DENNIS, JACK B.: "Virtual Memory, Processes, and Sharing in MULTICS," *CACM*, vol. 11, pp. 306–312, May 1968.

DALEY, ROBERT C., and NEUMANN, PETER G.: "A General Purpose File System for Secondary Storage," *Proc. FJCC*, pp. 213–229, 1965.

DARDNER, STEPHEN C., and HELLER, STEVEN B.: "Streamline Your Software Development," *Computer Decisions*, vol. 2, pp. 29–33, October 1970.

DAVIES, DONALD W., and BARBER, DEREK: *Communications Networks for Computers*. New York: Wiley, 1973.

DAVIES, P. M.: "Readings in Microprogramming," *IBM Systems Journal*, vol. 11, no. 1, pp. 16–40, 1972.

DENNING, PETER J.: "The Working Set Model for Program Behavior," *CACM*, vol. 11, pp. 323–333, May 1968.

DENNING, PETER J.: "Virtual Memory," *Computing Surveys*, vol. 2, pp. 153–189, September 1970.

DENNIS, J. B.: "The Design and Construction of Software Systems," in *Advanced Course on Software Engineering*, edited by Beckmann, Goos, and Kunzi. Berlin: Springer Verlag, 1972.

DIJKSTRA, E. W.: "Solution of a Problem in Concurrent Programming," *CACM*, vol. 8, p. 569, September 1965.

DIJKSTRA, E. W.: "GOTO Statement Considered Harmful," *CACM*, vol. 11, pp. 147–148, March 1968a.

DIJKSTRA, E. W.: The Structure of the "THE" Multiprogramming System," *CACM*, vol. 11, pp. 341–346, May 1968b.

DIJKSTRA, E. W.: "Co-operating Sequential Processes," in *Programming Languages*, edited by F. Genuys. New York: Academic Press, 1968c.

DIJKSTRA, E. W.: "Hierarchical Ordering of Sequential Processes," *Acta Informatica*, vol. 1, pp. 115–138, 1971.

DOLLHOFF, TERRY L.: "The Negative Aspects of Microprogramming," *Datamation*, vol. 20, pp. 64–66, July 1974.

DONOVAN, JOHN J.: *Systems Programming*. New York: McGraw-Hill, 1972.

FARBER, DAVID J.: "Networks: An Introduction," *Datamation*, vol. 18, pp. 36–39, April 1972.

FLORES, IVAN: *The Logic of Computer Arithmetic*. Englewood Cliffs, N.J.: Prentice-Hall, 1963.

FLORES, IVAN: "Lookahead Control in the IBM System 370 Model 165," *Computer*, vol. 7, pp. 24–38, Nov. 1974.

FLYNN, M. J., and MACLAREN, M. D.: "Microprogramming Revisited," *Proceedings ACM 22nd National Conference*, pp. 457–464, 1967.

FOSTER, CAXTON C.: *Computer Architecture*. New York: Van Nostrand Reinhold, 1970.

FOSTER, CAXTON C.: "The Next Three Generations," *Computer*, vol. 5, pp. 39–42, March–April 1972a.

FOSTER, CAXTON C.: "A View of Computer Architecture," *CACM*, vol. 15, pp. 557–565, July 1972b.

FOTHERINGHAM, J.: "Dynamic Storage Allocation in the Atlas Computer, Including an Automatic Use of a Backing Store," *CACM*, vol. 4, pp. 435–436, October 1961.

GARNER, HARVEY L.: "Number Systems and Arithmetic," in *Advances in Computers*. Vol. 6. Edited by F. Alt and M. Rubinoff. New York: Academic Press, pp. 131–194, 1965.

GEAR, C. WILLIAM: *Computer Organization and Programming*, 2nd ed. New York: McGraw-Hill, 1974.

GILDERSLEEVE, THOMAS R.: *Design of Sequential File Systems*. New York: Wiley-Interscience, 1971.

GOLDBERG, ROBERT P.: "Architecture of Virtual Machines," *Proc. NCC*, pp. 309–318, 1973.

GOLDBERG, ROBERT P. (Ed.): *Proceedings ACM SIGOPS—SIGARCH Workshop on Virtual Computer Systems*, ACM, New York, 1973.

GOLDBERG, ROBERT P., "Survey of Virtual Machine Research," *Computer*, pp. 34-43, June, 1974.

GOOS, G.: "Hierarchies," in *Advanced Course on Software Engineering*, edited by Beckmann, Goos, and Kunzi. Berlin: Springer Verlag, 1972.

GRAHAM, R.: "Use of High Level Languages for System Programming," *Project MAC Report TM–13*, Project MAC, MIT, September 1970.

HABERMANN, A. N.: "Prevention of System Deadlocks," *CACM*, vol. 12, pp. 373–377, July 1969.

HAMMING, RICHARD W.: "Error Detecting and Error Correcting Codes," *Bell System Technical Journal*, vol. 29, pp. 147–160, April 1950.

HANSEN, PER BRINCH: "Concurrent Programming Concepts," *Computing Surveys*, vol. 5, pp. 223–245, December 1973.

HANSEN, PER BRINCH: *Operating Systems Principles*. Englewood Cliffs, N.J.: Prentice-Hall, 1973.

HASSITT, A., LAGESCHULTE, J. W., and LYON, L. E.: "Implementation of a High Level Language Machine," *CACM*, vol. 16, pp. 199–212, April 1973.

HEART, F., KAHN, R., ORNSTEIN, S., CROWTHER, W., and WALDEN, D.: "The Interface Message Processor for the ARPA Computer Networks," *Proc. SJCC*, AFIPS Press, pp. 551–567, 1970.

HELLERMAN, HERBERT: *Digital Computer System Principles*. New York: McGraw-Hill, 1967.

HOLT, R. C.: "Some Deadlock Properties of Computer Systems," *Computing Surveys*, vol. 4, pp. 179–196, September 1972.

HUFFMAN, DAVID: "A Method for the Construction of Minimum Redundancy Codes," *Proc. IRE*, vol. 40, pp. 1098–1101, September 1952.

HUSSON, SAMIR S.: *Microprogramming: Principles and Practices*. Englewood Cliffs, N.J.: Prentice-Hall, 1970.

JUDD, D. R.: *Use of Files*. New York: American Elsevier, 1973.

KATZAN, HARRY, JR.: *Computer Organization and the System 370*. New York: Van Nostrand Reinhold, 1971.

KENT, WILLIAM: "Assembler-Language Macroprogramming. A Tutorial Oriented Toward the IBM 360," *Computing Surveys*, vol. 4, pp. 183–196, December 1969.

KNUTH, DONALD E.: *The Art of Computer Programming*. Vol. 1. *Fundamental Algorithms*, 2nd ed. Reading, Mass.: Addison-Wesley, 1974.

KNUTH, DONALD E.: *The Art of Computer Programming*. Vol. 2. *Seminumerical Algorithms*. Reading, Mass.: Addison-Wesley, 1971.

KNUTH, DONALD E.: *The Art of Computer Programming*. Vol. 3. *Sorting and Searching*. Reading, Mass.: Addison-Wesley, 1973.

LOESER, RUDOLF: "Some Performance Tests of Quicksort and Descendants," *CACM*, vol. 17, pp. 143–152, March 1974.

LORIN, HAROLD: *Parallelism in Hardware and Software: Real and Apparent Concurrency*. Englewood Cliffs, N.J.: Prentice-Hall, 1972.

LUKASIEWICZ, J.: *Aristotle's Syllogistic*, 2nd ed. New York: Oxford University Press, 1958.

LUM, V. Y., YVEN, P. S. T., and DODD, M.: "Key to Address Transformation Techniques—A Fundamental Performance Study on Large Existing Formatted Files," *CACM*, vol. 14, pp. 228–239, April 1971.

MADNICK, STUART E. and DONOVAN, JOHN J.: *Operating Systems*. New York: McGraw-Hill, 1974.

MALLACH, EFREM G.: "Emulator Architecture," *Computer* (to be published fourth quarter 1975).

MARTIN, JAMES: *Teleprocessing Network Organization.* Englewood Cliffs, N.J.: Prentice-Hall, 1970.

MARTIN, W.: "Sorting," *Computing Surveys*, vol. 3, pp. 147–174, December 1972.

MCCARTHY, J., CORBATÓ, F. J., and DAGGETT, N. M.: "The Linking Segment Subprogram Language and Linking Loader," *CACM*, vol. 6, pp. 391–395, June 1963.

MCDONELL, K. J., and MONTGOMERY, A. Y.: "The Design of Indexed Sequential Files," *Australian Computer Journal*, vol. 5, pp. 115–126, November 1973.

MORRIS, ROBERT: "Scatter Storage Techniques," *CACM*, vol. 11, pp. 38–44, January 1968.

NELSON, DONALD R. (Ed.): "Microprocessor Architecture," *Computer*, vol 7., pp.1–72, July 1974.

ORGANICK, ELLIOT: *The MULTICS System.* Cambridge, Mass.: MIT Press, 1972.

PARMELEE, R. P., PETERSON, T. I., TILLMAN, C. C., and HATFIELD, D. J.: "Virtual Storage and Machine Concepts," *IBM System Journal*, vol. 11, pp. 99–130, 1972.

PETERSON, W. W.: *Error Correcting Codes.* Cambridge, Mass.: MIT Press, 1961.

POPEK, GERALD J., and GOLDBERG, ROBERT P.: "Formal Requirements for Virtualizable Third Generation Architectures," *CACM*, vol. 17, pp. 412–421, July 1974.

PYKE, T. N., JR., and BLANC, R. P.: "Computer Networking Technology—A State of the Art Review," *Computer*, vol. 6, pp. 13–19, August 1973.

PRESSER, LEON, and WHITE, JOHN R.: "Linkers and Loaders," *Computing Surveys*, vol. 4, pp. 150–167, September 1972.

PRICE, C. E.: "Table Lookup Techniques," *Computing Surveys*, vol. 3, pp. 49–65, June 1971.

RANDELL, B., and KUEHNER, C. J.: "Dynamic Storage Allocation Systems," *CACM*, vol. 11, 297–306, May 1968.

REITWIESNER, GEORGE W.: "Binary Arithmetic," in *Advances in Computers.* Vol. 1. Edited by F. Alt. New York: Academic Press, pp. 232–308, 1960.

RICH, ROBERT P.: *Internal Sorting Methods Illustrated with PL/1 Programs.* Englewood Cliffs, N.J.: Prentice-Hall, 1972.

ROBERTS, DAVID C.: "File Organization Techniques," in *Advances in Computers.* Vol. 12. Edited by M. Rubinoff. New York: Academic Press, pp. 115–174, 1972.

ROBERTS, L., and WESSLER, B.: "Computer Network Development to Achieve Resource Sharing," *Proc. SJCC 1970*, pp. 543–549, 1970.

ROSIN, ROBERT F.: "Contemporary Concepts of Microprogramming and Emulation," *Computing Surveys*, vol. 1, pp. 197–212, December 1969.

ROSIN, ROBERT F., FRIEDER, GIDEON, and ECKHOUSE, RICHARD H., JR.: "An Environment for Research in Microprogramming and Emulation," *CACM*, vol. 15, pp. 748–760, August 1972.

ROSIN, ROBERT F.: "The Significance of Microprogramming," *SIGMICRO Newsletter*, vol. 4, pp. 24–39, January 1974.

RUSTIN, RANDALL (Ed.): *Computer Networks.* Englewood Cliffs, N.J.: Prentice-Hall, 1972.

SALTZER, JEROME H.: "A Simple Linear Model of Demand Paging Performance," *CACM*, vol. 17, pp. 181–186, April 1974.

SANDERS, RAY W.: "Networking, an Overview," *Datamation*, vol. 20, pp. 54–55, March 1974.

SHAW, A.: *The Logical Design of Operating Systems*. Englewood Cliffs, N.J.: Prentice-Hall, 1974.

SHELL, D. L.: "A High Speed Sorting Procedure," *CACM*, vol. 2, pp. 30–32, January 1959.

SIGPLAN: "Proceedings of a Symposium on High Level Language Computer Architecture," *SIGPLAN Notices*, vol. 8, November 1973.

STEIN, MARVIN L., and MUNRO, WILLIAM D.: *Introduction to Machine Arithmetic*. Reading, Mass.: Addison-Wesley, 1971.

STONE, HAROLD S.: *Introduction to Computer Organization and Data Structures*. New York: McGraw-Hill, 1972.

STRACHEY, C.: "A General Purpose Macrogenerator," *Computer Journal*, vol. 8, pp. 225–241, March 1965.

SUMNER, F. H.: "MU5—An Assessment of the Design," *Proc. IFIP Congress 74*, pp. 133–136, 1974.

THORNTON, J. E.: *Design of a Computer: The Control Data 6600*. Glenview, Ill.: Scott, Foresman, 1970.

TSICHRITZIS, DIONYSIOS, and BERNSTEIN, PHILIP: *Operating Systems*. New York: Academic Press, 1974.

TUCKER, S. G.: "Microprogram Control for System/360," *IBM Systems Journal*, vol. 6, no. 4, pp. 222–241, 1967.

VAN DER POEL, W. L.: "The Software Crisis, Some Thoughts and Outlooks," *Proc. IFIP Congress 68*, pp. 334–339, 1968.

VAN WIJNGAARDEN, A.: "Numerical Analysis as an Independent Science," *BIT*, vol. 6, no. 1, pp. 66–81, 1966.

WAITE, W. M.: "The Mobile Programming System: STAGE 2," *CACM*, vol. 13, pp. 415–421, July 1970.

WEBER, H.: "A Microprogrammed Implementation of Euler on the IBM 360/30," *CACM*, vol. 10, pp. 549–558, September 1967.

WEGNER, PETER: *Programming Languages, Information, Structures, and Machine Organization*. New York: McGraw-Hill, 1968.

WESSLER, BARRY D., and HOVEY, RICHARD B.: "Public Packet-Switched Networks," *Datamation*, vol. 20, pp. 85–87, July 1974.

WILKES, M. V.: "The Growth of Interest in Microprogramming: A Literature Survey," *Computing Surveys*, vol. 1, pp. 139–145, September 1969.

WILKES, M. V.: *Time-Sharing Computer Systems*. 2nd ed. New York: American Elsevier, 1972.

WILNER, WAYNE: "Design of the Burroughs B1700," *Proc. FJCC*, pp. 489–497, 1972.

WIRTH, NIKLAUS: "On Multiprogramming, Machine Coding and Computer Organization," *CACM*, vol. 12, pp. 489–498, September 1969.

ZURCHER, F. W., and RANDELL B.: "Iterative Multi-level Modeling—A Methodology for Computer System Design," *Proc. IFIP Congress 68*, pp. 867–871, 1969.

APPENDIX

A FINITE-PRECISION ARITHMETIC AND BINARY NUMBERS

The arithmetic used by computers differs in some ways from the arithmetic used by people. The most important difference is that computers perform operations on numbers whose precision is finite and fixed. Another difference is that most computers use the binary rather than the decimal system for representing numbers. These topics are the subject of this appendix.

A.1. FINITE-PRECISION NUMBERS

While doing arithmetic, one usually gives little thought to the question of how many decimal digits it takes to represent a number. Physicists can calculate that there are 10^{78} electrons in the universe without being bothered by the fact that it requires 79 decimal digits to write that number out in full. If someone who is calculating the value of a function with pencil and paper needs the answer to six significant digits, he simply keeps intermediate results to seven, or eight, or however many are needed. There is never any problem of the paper not being wide enough for seven-digit numbers.

With computers, matters are quite different. On most computers, the amount of memory available for storing a number is fixed at the time that the computer is designed. With a certain amount of effort, the programmer can represent numbers two, or three, or even many times larger than this fixed amount, but doing so does not change the nature of this difficulty. The finite nature of the computer forces us to deal only with numbers that can be represented in a fixed number of digits. We call such numbers **finite-precision numbers**.

In order to study properties of finite-precision numbers, let us examine the set of positive integers representable by three decimal digits, with no decimal point and no sign. This set has exactly 1000 members: 000, 001, 002, 003, . . . , 999. In this system it is impossible to express several important sets of numbers, such as

1. Numbers larger than 999.
2. Negative numbers.
3. Fractions.
4. Irrational numbers.
5. Complex numbers.

One important property of arithmetic on the set of all integers is closure with respect to the operations of addition, subtraction, and multiplication. In other words, for every pair of integers i and j, $i + j$, $i - j$, and $i \times j$ are also integers. The set of integers is not closed with respect to division because there exist values of i and j for which i/j is not expressible as an integer—for example, 7/2 and 1/0.

Finite-precision numbers, unfortunately, are not closed with respect to any of these four basic operations, as shown below, using three-digit decimal numbers as an example:

```
600 + 600 = 1200    (too large)
503 - 505 =   -2    (negative)
050 × 050 = 2500    (too large)
007 / 002 = 3.5     (not an integer)
```

The violations can be divided into two mutually exclusive classes: operations whose result is larger than the largest number in the set (overflow error) or smaller than the smallest number in the set, and operations whose result is neither too large nor too small but is simply not a member of the set. Of the four violations above, the first three are examples of the former, and the fourth is an example of the latter.

Because computers have finite memories and therefore must, of necessity, perform arithmetic on finite-precision numbers, the results of certain calculations will be, from the point of classical mathematics, just plain wrong. A calculating device that gives the wrong answer even though it is in perfect working condition may appear strange at first, but is a logical consequence of its finite nature. Some computers have special hardware that detects overflow errors.

The algebra of finite-precision numbers is different from normal algebra. As an example, consider the associative law:

$$a + (b - c) = (a + b) - c$$

Let us evaluate both sides for $a = 700$, $b = 400$, $c = 300$. To compute the left-hand side, first calculate $(b - c)$, which is 100, and then add this amount to a, yielding 800. To compute the right-hand side, first calculate $(a + b)$, which gives an overflow in the finite arithmetic of three-digit integers. The result may depend on the machine

being used, but it will not be 1100. Subtracting 300 from some number other than 1100 will not yield 800. The associative law does not hold. The order of operations is important.

As another example, consider the distributive law:

$$a \times (b - c) = a \times b - a \times c$$

Let us evaluate both sides for $a = 5$, $b = 210$, $c = 195$. The left-hand side is 5×15, which yields 75. The right-hand side is not 75 because $a \times b$ overflows.

Judging from these examples, one might conclude that although computers are general-purpose devices, their finite nature renders them especially unsuitable for doing arithmetic. This conclusion is, of course, not true, but it does serve to illustrate the importance of understanding how computers work and what limitations they have.

A.2. RADIX NUMBER SYSTEMS

An ordinary decimal number with which everyone is familiar consists of a string of decimal digits and, possibly, a decimal point. The general form and its usual interpretation are shown in Fig. A-1. The choice of 10 as the base for exponentiation, called the **radix**, is made because we are using decimal, or base 10, numbers. When dealing with computers, it is frequently convenient to use radices other than 10. The most important radices are 2, 8, and 16. The number systems based on these radices are called **binary**, **octal**, and **hexadecimal**, respectively.

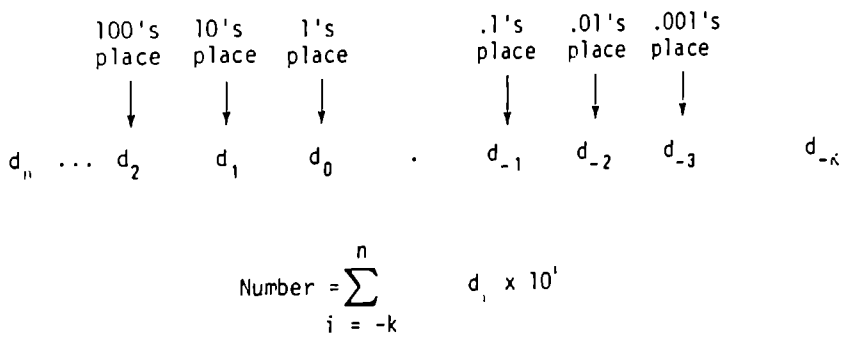


Fig. A-1. The general form of a decimal number.

A radix k number system requires k different symbols to represent the digits 0 to $k - 1$. Decimal numbers are built up from the 10 decimal digits

0 1 2 3 4 5 6 7 8 9.

Binary numbers are built up from the two binary digits

0 and 1

Octal numbers are built up from the eight octal digits

0 1 2 3 4 5 6 7

For hexadecimal numbers, 16 digits are needed. Thus six new symbols are required. It is conventional to use the upper-case letters A through F for the six digits following 9. Hexadecimal numbers are then built up from the digits

0 1 2 3 4 5 6 7 8 9 A B C D E F

The expression "binary digit," meaning a 1 or a 0, is usually referred to as a **bit**. Figure A-2 shows the decimal number 1977 expressed in binary, octal, and hexadecimal form. The number 7B9 is obviously hexadecimal, since the symbol B can only occur in hexadecimal numbers. However, the number 111 might be in any of the four number systems discussed. To avoid ambiguity, people use a subscript of 2, 8, 10, or 16 to indicate the radix when it is not obvious from the context. Figure A-3 shows some numbers expressed in all four systems.

Binary

$$\begin{array}{cccccccccccc} 1 & & 1 & & 1 & & 1 & & 0 & & 1 & & 1 & & 1 & & 0 & & & & 1 \\ 1\times 2^{10} & + & 1\times 2^{9} & + & 1\times 2^{8} & + & 1\times 2^{7} & + & 1\times 2^{6} & + & 1\times 2^{5} & + & 1\times 2^{4} & + & 1\times 2^{3} & + & 1\times 2^{2} & + & 1\times 2^{1} & + & 1\times 2^{0} \\ 1024 & + & 512 & + & 256 & + & 128 & + & 0 & + & 32 & + & 16 & + & 8 & + & 0 & + & 0 & + & 1 \end{array}$$

Octal

$$\begin{array}{cccccccc} 3 & & 6 & & 7 & & 1 \\ 3\times 8^{3} & + & 6\times 8^{2} & + & 7\times 8^{1} & + & 1\times 8^{0} \\ 1536 & + & 384 & + & 56 & + & 1 \end{array}$$

Hexadecimal

$$\begin{array}{cccccc} 7 & & B & & 9 \\ 7\times 16^{2} & + & 11\times 16^{1} & + & 9\times 16^{0} \\ 1792 & + & 176 & + & 9 \end{array}$$

Fig. A-2. The number 1977 in binary, octal, and hexadecimal.

A.3. CONVERSION FROM ONE RADIX TO ANOTHER

Conversion between octal or hexadecimal numbers and binary numbers is very easy. To convert a binary number to octal, divide it into groups of three bits, with the three bits immediately to the left (or right) of the decimal point (often called a binary point) forming one group, the three bits immediately to their left another group, and so on. Each group of three bits can be directly converted to a single octal digit, 0 to 7, according to the conversion given in the first lines of Fig. A-3. It may be necessary to add one or two leading or trailing zeros to fill out a group to three bits. Conversion from octal to binary is equally easy. Each octal digit is simply replaced by the equivalent three-bit binary number. Hexadecimal-binary conversion is essen-

Decimal	Binary	Octal	Hexadecimal
0	0	0	0
1	1	1	1
2	10	2	2
3	11	3	3
4	100	4	4
5	101	5	5
6	110	6	6
7	111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F
16	10000	20	10
17	10001	21	11
18	10010	22	12
19	10011	23	13
20	10100	24	14
30	11110	36	1E
40	101000	50	28
50	110010	62	32
60	111100	74	3C
70	1000110	106	46
80	1010000	120	50
90	1011010	132	5A
100	1100100	144	64
200	11001000	310	C8
300	100101100	454	12C
400	110010000	620	190
500	111110100	764	1F4
600	1001011000	1130	258
700	1010111100	1274	2BC
800	1100100000	1440	320
900	1110000100	1604	384
1000	1111101000	1750	3E8
2989	101110101101	5655	BAD

Fig. A-3. Decimal numbers and their binary, octal, and hexadecimal equivalents.

tially the same as octal binary except that each hexadecimal digit corresponds to a group of four bits instead of three. Figure A-4 gives some examples of conversions.

Conversion of decimal numbers to binary can be done in two different ways. The first method follows directly from the definition of binary numbers. The largest power of 2 smaller than the number is subtracted from the number. The process is then repeated on the difference. Once the number has been decomposed into powers of 2, the binary number can be assembled with ones in the bit positions corresponding to powers of 2 used in the decomposition, and zeros elsewhere.

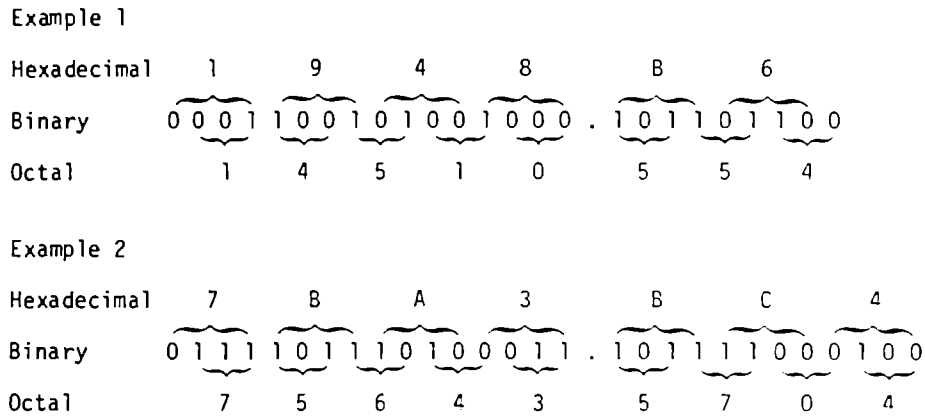


Fig. A-4. Examples of octal-binary and hexadecimal-binary conversion.

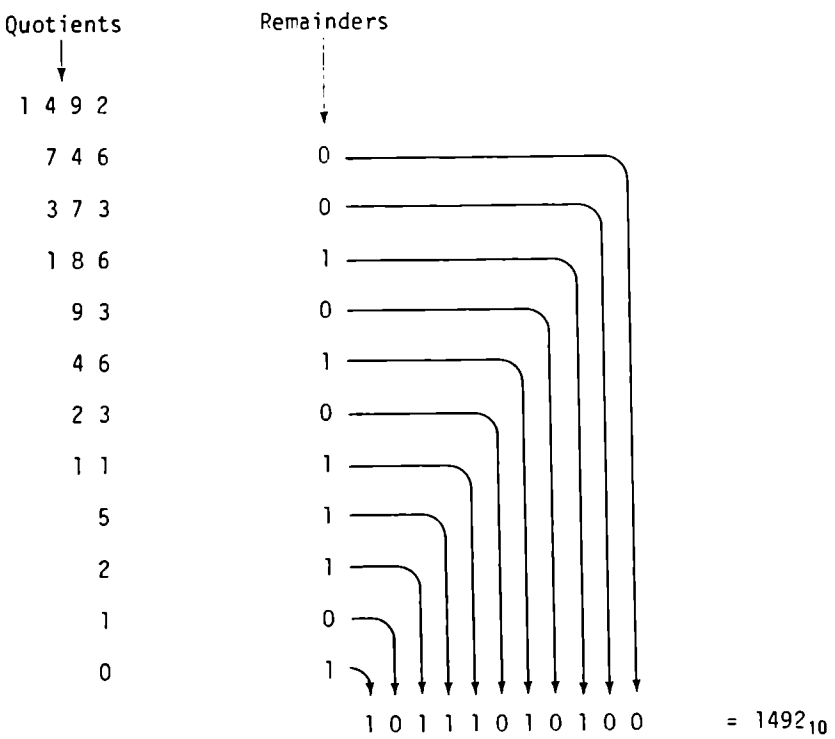


Fig. A-5. Conversion of the decimal number 1492 to binary by successive halving, starting at the top and working downward. For example, 93 divided by 2 yields a quotient of 46 and a remainder of 1, written on the line below it.

The other method (for integers only) consists of dividing the number by 2. The quotient is written directly beneath the original number, and the remainder, 0 or 1, is written next to the quotient. The quotient is then considered and the process repeated until the number 0 has been reached. The result of this process will be two columns of numbers, the quotients and the remainders. The binary number can now be read directly from the remainder column, starting at the bottom. Figure A-5 gives an example of decimal-to-binary conversion.

There are also two ways to convert binary integers to decimal. One method consists of summing up the powers of 2 corresponding to the one bits in the number. For example, 10110 is $2^4 + 2^2 + 2^1 = 16 + 4 + 2 = 22$. In the other method, the binary number is written vertically, one bit per line, with the leftmost bit on the bottom. The bottom line is called line 1, the one above it line 2, and so forth. The decimal number will be built up in a parallel column next to the binary number. Begin by writing a 1 on line 1. The entry on line N consists of two times the entry on line N − 1 plus the bit on line N (either 0 or 1). The entry on the top line is the answer. Figure A-6 gives an example of this method of binary-to-decimal conversion.

Decimal-octal and decimal-hexadecimal conversion can be accomplished either by first converting to binary and then to the desired system or by subtracting off powers of octal or hexadecimal numbers.

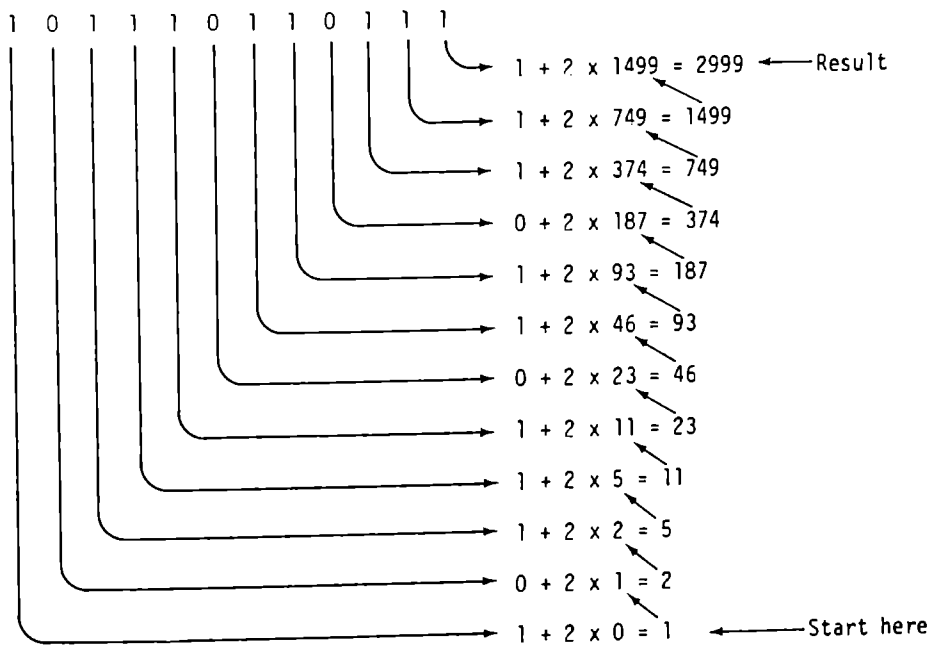


Fig. A-6. Conversion of the binary number 101110110111 to decimal by successive doubling, starting at the bottom. Each line is formed by doubling the one below it and adding the corresponding bit. For example, 749 is twice 374 plus the 1 bit on the same line as 749.

A.4. NEGATIVE BINARY NUMBERS

Four systems for representing negative numbers have been used in digital computers. The first one is called **signed magnitude**. In this system the leftmost bit is the sign bit (0 is + and 1 is −), and the remaining bits hold the absolute magnitude of the number.

The second system, called **1's complement**, also has a sign bit with 0 used for plus and 1 for minus. To negate a number, replace each 1 by a 0 and each 0 by a 1. This holds for the sign bit as well.

The third system, called **2's complement**, also has a sign bit that is 0 for plus and 1 for minus. Negating a number is a two-step process. First, each 1 is replaced by a 0 and each 0 by a 1, just as in 1's complement. Second, 1 is added to the result. Binary addition is the same as decimal addition except that a carry is generated if the sum is greater than 2 rather than greater than 10. For example, converting 6 to 2's complement is done in two steps:

```
00000110    (6)
11111001    (−6 in 1's complement)
       1    (add 1)
--------
11111010    (−6 in 2's complement)
```

If a carry occurs from the leftmost bit, it is thrown away.

The fourth system, which for M-bit numbers is called **excess** 2^{M-1}, represents a number by storing it as the sum of itself and 2^{M-1}. For example, for eight-bit numbers, M = 8, the system is called excess 128 and a number is stored as its true value plus 128. Therefore −3 becomes −3 + 128 = 125, and −3 is represented by the eight-bit binary number for 125 (01111101). The numbers from −128 to +127 map onto 0 to 255, all of which are expressible as an eight-bit positive integer. Interestingly enough, this system is identical to 2's complement with the sign bit reversed. Figure A-7 gives examples of all four systems.

Both signed magnitude and 1's complement have two representations for zero, a plus zero and a minus zero. This situation is highly undesirable. The 2's complement system does not have this problem, since the 2's complement of plus zero is also plus zero. The 2's complement system does have a singularity, however. The bit pattern consisting of a 1 followed by all 0's is its own complement. The result is to make the range of positive and negative numbers unsymmetric; there is one negative number with no positive counterpart. The excess notation has the same idiosyncracy.

The reason for these problems is very simple. Ideally, we would like an encoding system with two properties:

1. There is only one representation for zero.
2. There are exactly as many positive numbers as negative numbers.

The basic problem is that any set of numbers with as many positive as negative numbers and only one zero has an odd number of members, whereas N bits allow an even number of bit patterns. There will always be either one bit pattern too many

N (decimal)	N (binary)	-N (signed mag.)	-N (1's compl.)	-N (2's compl.)	-N (excess 128)
0	00000000	10000000	11111111	00000000	10000000
1	00000001	10000001	11111110	11111111	01111111
2	00000010	10000010	11111101	11111110	01111110
3	00000011	10000011	11111100	11111101	01111101
4	00000100	10000100	11111011	11111100	01111100
5	00000101	10000101	11111010	11111011	01111011
6	00000110	10000110	11111001	11111010	01111010
7	00000111	10000111	11111000	11111001	01111001
8	00001000	10001000	11110111	11111000	01111000
9	00001001	10001001	11110110	11110111	01110111
10	00001010	10001010	11110101	11110110	01110110
11	00001011	10001011	11110100	11110101	01110101
12	00001100	10001100	11110011	11110100	01110100
13	00001101	10001101	11110010	11110011	01110011
14	00001110	10001110	11110001	11110010	01110010
15	00001111	10001111	11110000	11110001	01110001
16	00010000	10010000	11101111	11110000	01110000
17	00010001	10010001	11101110	11101111	01101111
18	00010010	10010010	11101101	11101110	01101110
19	00010011	10010011	11101100	11101101	01101101
20	00010100	10010100	11101011	11101100	01101100
30	00011110	10011110	11100001	11100010	01100010
40	00101000	10101000	11010111	11011000	01011000
50	00110010	10110010	11001101	11001110	01001110
60	00111100	10111100	11000011	11000100	01000100
70	01000110	11000110	10111001	10111010	00111010
80	01010000	11010000	10101111	10110000	00110000
90	01011010	11011010	10100101	10100110	00100110
100	01100100	11100100	10011011	10011100	00011100
127	01111111	11111111	10000000	10000001	00000001
128	--------	--------	--------	10000000	00000000

Figure A-7. Negative numbers in four systems (8-bit numbers).

or one bit pattern too few, no matter what representation is chosen. This extra bit pattern can be used for −0, or a large negative number, or something else, but it will always be a nuisance. Alternatively, a clever computer designer might use this extra bit pattern to represent "undefined," in order to trap the use of uninitialized variables.

A.5. BINARY ARITHMETIC

The addition table for binary numbers is given below.

Addend	0	0	1	1
Augend	+0	+1	+0	+1
Sum	0	1	1	0
Carry	0	0	0	1

Two binary numbers can be added, starting at the rightmost bit and adding the corresponding bits in the addend and the augend. If a carry is generated, it is carried one position to the left, just as in decimal arithmetic. In 1's complement arithmetic, a carry generated by the addition of the leftmost bits is added to the rightmost bit. This process is called an end-around carry. In 2's complement arithmetic, a carry generated by the addition of the leftmost bits is merely thrown away. An example of addition in 1's complement and 2's complement form is shown below.

Decimal	1's complement	2's complement
10	00001010	00001010
+(−3)	11111100	11111101
	1 00000110	1 00000111
	carry 1	discarded
	00000111	

If the addend and the augend are of opposite signs, overflow error cannot occur. If they are of the same sign, and the result is of the opposite sign, overflow error has occurred and the answer is wrong. In both 1's and 2's complement arithmetic, overflow occurs if and only if the carry into the sign bit differs from the carry out of the sign bit.

PROBLEMS

1. Convert the following decimal numbers to binary: 1984, 2001, 8192.
2. What is 1001101001 (binary) in decimal? In octal? In hexadecimal?
3. Which of the following are valid hexadecimal numbers: BED, CAB, DEAD, DECADE, CEDE, BAG, DAD?
4. Express the decimal number 100 in all radices from 2 to 9.
5. How many different positive integers can be expressed in k digits using radix r numbers?
6. Does the commutative law for addition ($A + B = B + A$) hold for finite-precision arithmetic?
7. Most people can only count to 10 on their fingers; however, computer scientists can do better. If you regard each finger as one binary bit, with finger extended = 1 and finger touching palm = 0, how high can you count using both hands? If your left thumb is used as sign bit for 2's complement numbers, what is the range of expressible numbers?
8. Perform the following calculations on eight bit 2's complement numbers.

00101101	11111111	00000000	11110111
+01101111	+11111111	−11111111	−11110111

9. Perform the same calculations as in Problem 8 but treat the numbers as 1's complement instead of 2's complement.

10. Consider the following addition problems for three-bit binary numbers in 2's complement. For each sum, tell
    a. whether the sign bit of the result is 1.
    b. whether the low-order three bits of the result are 0.
    c. whether an overflow occurred.

000	000	111	100	100
001	111	110	111	100

11. Consider a radix 2 number system with 3 possibilities for each digit: -1, 0, $+1$.
    a. prove that every positive integer has an infinite number of representations.
    b. give 5 representations for 13 (decimal).

12. Signed decimal numbers consisting of N digits can be represented in N + 1 digits without a sign. Positive numbers have 0 as the leftmost digit. Negative numbers are formed by subtracting each digit from 9. Thus the negative of 014725 is 985274. Such numbers are called 9's complement numbers and are analogous to 1's complement binary numbers. Express the following as three-digit 9's complement numbers: 6, -2, 100, -14, -1, 0.

13. Perform the following additions, using 9's complement numbers.

0001	0001	9997	9241
+9999	+9998	+9996	+0802

14. 10's complement numbers are analogous to 2's complement numbers. A 10's complement negative number is formed by adding 1 to the corresponding 9's complement number, ignoring the carry. What is the rule for 10's complement addition?

15. Construct the multiplication tables for radix 3 numbers.

16. Multiply the numbers 0111 and 0011 in binary.

APPENDIX

B FLOATING-POINT NUMBERS

In many calculations the range of numbers used is very large. For example, a calculation in astronomy might involve the mass of the electron, 9×10^{-28} grams, and the mass of the sun, 2×10^{33} grams, a range exceeding 10^{60}. These numbers could be represented by

```
000000000000000000000000000000000.000000000000000000000000009
200000000000000000000000000000000.000000000000000000000000000
```

and all calculations could be carried out keeping 34 digits to the left of the decimal point and 28 places to the right of it. Doing so would allow 62 significant digits in the results. On a binary computer, multiple-precision arithmetic could be used to provide enough significance. However, the mass of the sun is not even known accurately to five significant digits, let alone 62. In fact, very few measurements of any kind can (or need) be made accurately to 62 significant digits. Although it would be possible to keep all intermediate results to 62 significant digits and then throw away 50 or 60 of them before printing the final results, doing so is wasteful of both CPU time and memory. What is needed is a system for representing numbers in which the range of expressible numbers is independent of the number of significant digits.

One way of separating the range from the precision is to express numbers in the familiar scientific notation

$$n = f \times 10^e$$

where f is called the **fraction**, or **mantissa**, and e is a positive or negative integer called the **exponent**. The computer version of this notation is called **floating point.**

Some examples of numbers expressed in this form are

$$\begin{aligned} 3.14 &= 0.314 \times 10^{1} = 3.14 \times 10^{0} \\ 0.000001 &= 0.1 \times 10^{-5} = 1.0 \times 10^{-6} \\ 1941 &= 0.1941 \times 10^{4} = 1.941 \times 10^{3} \end{aligned}$$

Note that the range is effectively determined by the number of digits in the exponent and that the precision is determined by the number of digits in the fraction. Because there is more than one way to represent a given number, one form is usually chosen as the standard. In order to investigate the properties of this method of representing numbers, consider a representation, R, with a signed three-digit fraction in the range $1 < f \leq 0.1$ or zero and a signed two-digit exponent. These numbers range in magnitude from $+0.100 \times 10^{-99}$ to $+0.999 \times 10^{+99}$, a span of nearly 199 orders of magnitude, yet only five digits and two signs are needed to store a number.

Floating-point numbers can be used to model the real-number system of mathematics, although there are some important differences. Figure B-1 gives a grossly exaggerated schematic of the real-number line. The real line is divided up into seven regions:

1. Large negative numbers less than -0.999×10^{99}.
2. Negative numbers between -0.999×10^{99} and -0.100×10^{-99}.
3. Small negative numbers with magnitudes less than 0.100×10^{-99}.
4. Zero.
5. Small positive numbers with magnitudes less than 0.100×10^{-99}.
6. Positive numbers between 0.100×10^{-99} and 0.999×10^{99}.
7. Large positive numbers greater than 0.999×10^{99}.

One major difference between the set of numbers representable with three fraction and two exponent digits and the real numbers is that the former cannot be used to express any numbers in regions 1, 3, 5, or 7. If the result of an arithmetic operation yields a number in regions 1 or 7—for example, $10^{60} \times 10^{60} = 10^{120}$—an overflow error will occur, and the answer will be incorrect. The reason is due to the finite nature of the representation for numbers and is unavoidable. Similarly, a result in region 3 or 5 cannot be expressed either. This is called underflow error. Underflow error is less serious than overflow error, because 0 is often a satisfactory approximation to numbers in regions 3 and 5. A bank balance of 10^{-102} dollars is hardly better than a bank balance of 0.

Another important difference between floating-point numbers and the real numbers is their density. Between any two real numbers, X and Y, is another real number, no matter how close X is to Y. This property comes from the fact that for any distinct real numbers, X and Y, $Z = (X + Y)/2$ is a real number between them. The real numbers form a continuum.

Floating-point numbers, in contrast, do not form a continuum. There are 179,100 positive numbers that can be expressed in the five-digit 2 sign system used above, 179,100 negative numbers and 0 (which can be expressed in many ways), for a total

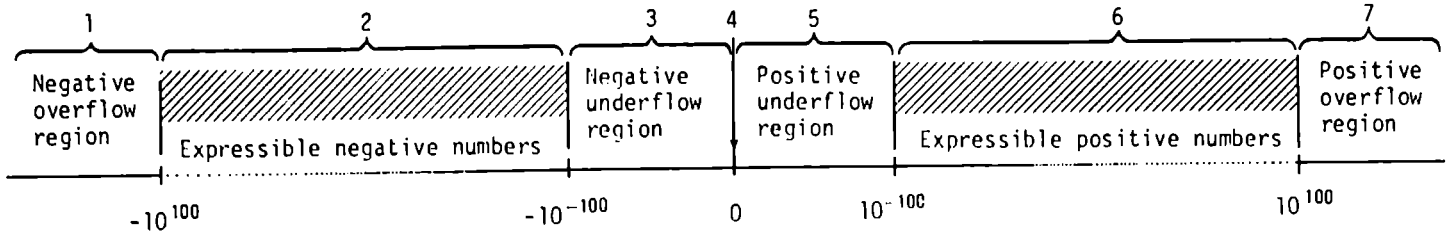


Fig. B-1. The real number line can be divided into seven regions.

of 358,201 numbers. Of the infinite number of real numbers between 10^{-100} and $+0.999 \times 10^{99}$, only 358,201 of them can be specified by this notation. They are symbolized by the dots in Fig. B-1. It is quite possible for the result of a calculation to be one of the other numbers, even though it is in region 2 or 6. For example, $+0.100 \times 10^{3}$ divided by 3 cannot be expressed *exactly* in our system of representation. If the result of a calculation cannot be expressed, in the number representation being used, the obvious thing to do is to use the nearest number that can be expressed. This process is called **rounding**.

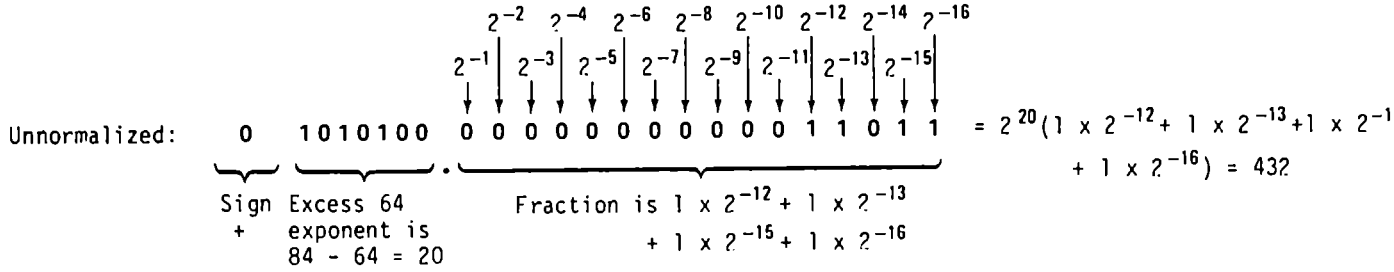
It is interesting to note that the spacing between adjacent expressible numbers is not constant throughout regions 2 or 6. The separation between $+0.998 \times 10^{99}$ and $+0.999 \times 10^{99}$ is vastly more than the separation between $+0.998 \times 10^{0}$ and $+0.999 \times 10^{0}$. However, when the separation between a number and its successor is expressed as a percentage of that number, there is no systematic variation throughout regions 2 or 6. In other words, the **relative error** introduced by rounding is the same for small numbers as large numbers.

Although the preceding discussion was in terms of a representation system with a three-digit fraction and a two-digit exponent, the conclusions drawn are valid for other representation systems as well. Changing the number of digits in the fraction

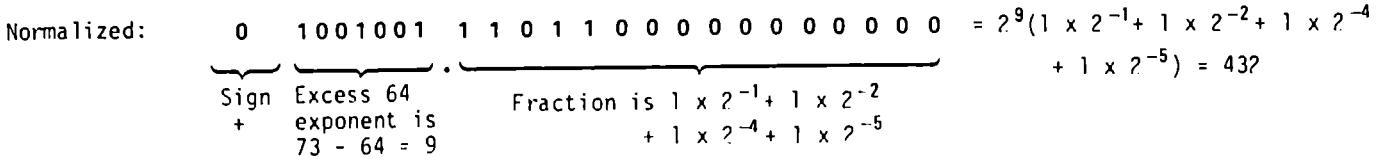
Digits in fraction	Digits in exponent	Lower	Upper
3	1	10^{-12}	10^{9}
3	2	10^{-102}	10^{99}
3	3	10^{-1002}	10^{999}
3	4	10^{-10002}	10^{9999}
4	1	10^{-13}	10^{9}
4	2	10^{-103}	10^{99}
4	3	10^{-1003}	10^{999}
4	4	10^{-10003}	10^{9999}
5	1	10^{-14}	10^{9}
5	2	10^{-104}	10^{99}
5	3	10^{-1004}	10^{999}
5	4	10^{-10004}	10^{9999}
10	3	10^{-1009}	10^{999}
20	3	10^{-1019}	10^{999}

Fig. B-2. The approximate lower and upper bounds of expressible (unnormalized) floating point decimal numbers.

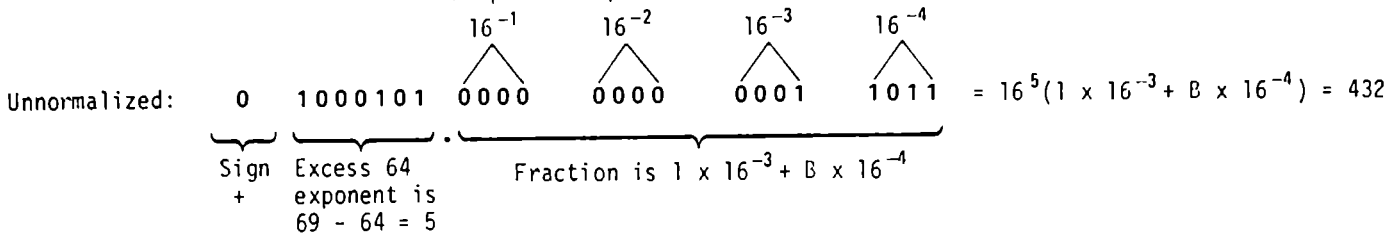
Example 1: Exponentiation to the base 2



To normalize, shift the fraction left 11 bits and subtract 11 from the exponent.


Normalized: 0 1001001 1 1 0 1 1 0 0 0 0 0 0 0 0 0 0
$= 2^{9}(1 \times 2^{-1} + 1 \times 2^{-2} + 1 \times 2^{-4} + 1 \times 2^{-5}) = 432$
Sign + Excess 64 exponent is 73 - 64 = 9
Fraction is $1 \times 2^{-1} + 1 \times 2^{-2} + 1 \times 2^{-4} + 1 \times 2^{-5}$

Example 2: Exponentiation to the base 16


16^{-1} 16^{-2} 16^{-3} 16^{-4}
Unnormalized: 0 1000101 0000 0000 0001 1011
$= 16^{5}(1 \times 16^{-3} + B \times 16^{-4}) = 432$
Sign + Excess 64 exponent is 69 - 64 = 5
Fraction is $1 \times 16^{-3} + B \times 16^{-4}$

To normalize, shift the fraction left 2 hexadecimal digits, and subtract 2 from the exponent.

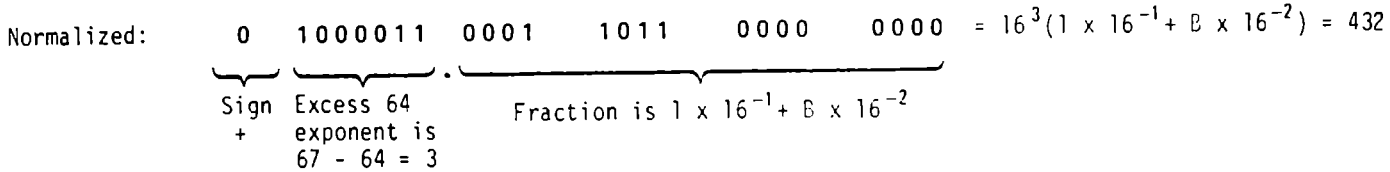
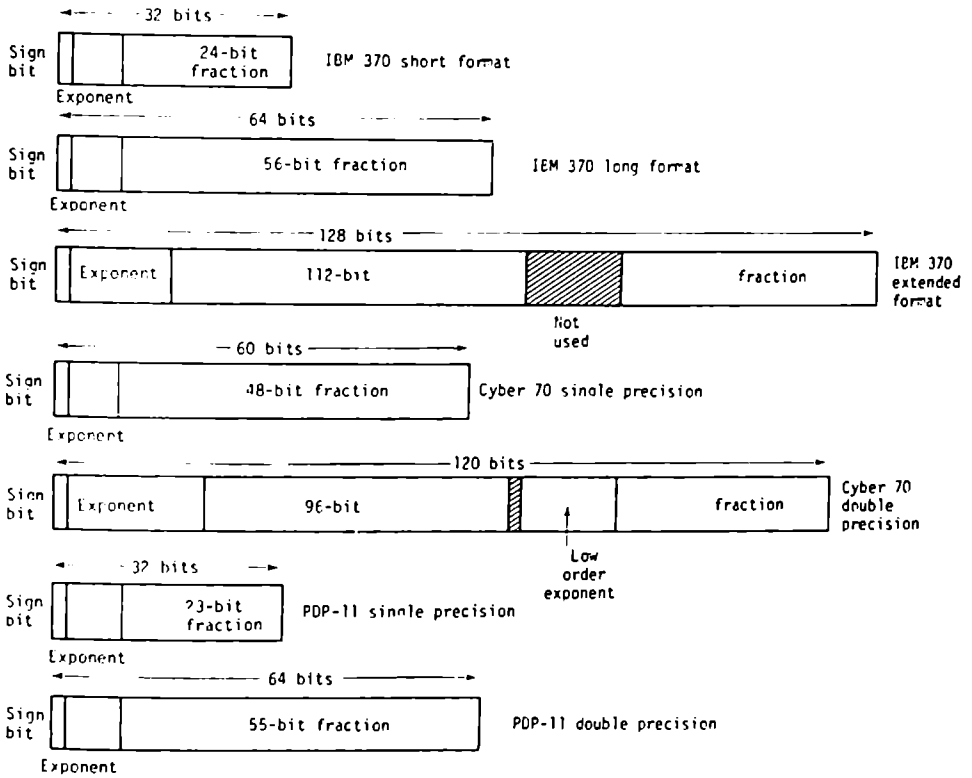

Normalized: 0 1000011 0001 1011 0000 0000
$= 16^{3}(1 \times 16^{-1} + B \times 16^{-2}) = 432$
Sign + Excess 64 exponent is 67 - 64 = 3
Fraction is $1 \times 16^{-1} + B \times 16^{-2}$

Fig. B-3. Examples of normalizing floating point numbers.

or exponent merely shifts the boundaries of regions 2 and 6 and changes the number of expressible points in them. Increasing the number of digits in the fraction increases the density of points and therefore improves the accuracy of approximations. Increasing the number of digits in the exponent increases the size of regions 2 and 6 by shrinking regions 1, 3, 5, and 7. Figure B-2 shows the approximate boundaries of region 6 for floating-point decimal numbers for various sizes of fraction and exponent.

A variation of this representation is used in computers. For efficiency, exponen-



Computer	Radix of exponentiation	Bits in the exponent	Number system for exponent	Number system for fraction	A number is normalized if	Location of the radix point	Approximate largest number
IBM 370	16	7	Excess 64	Signed magnitude	High order hex digit non zero	To the left of the fraction	10^{76}
Cyber 70	?	11	Excess 1024	One's complement	High order bit non zero	To the right of the fraction	10^{322}
PDP-11	?	8	Excess 128	Signed magnitude	Always	To the left of the fraction	10^{38}

Fig. B-4. Floating point number formats used on the IBM 370, CDC Cyber 70, and DEC PDP-11. All PDP-11 floating point numbers are normalized, so the high order fraction bit is always a 1, and is not stored. This allows 24 (or 56) bits of precision to be encoded in a 23 (or 55) bit field.

tiation is usually to base 2, 4, 8, or 16, rather than 10, in which case the fraction consists of a string of binary, base 4, octal, or hexadecimal digits. If the leftmost of these digits is zero, all the digits can be shifted one place to the left and the exponent decreased by 1, without changing the value of the number (barring underflow). A fraction with a nonzero leftmost digit is said to be **normalized**.

Normalized numbers are generally preferable to nonnormalized numbers, because there is only one normalized form, whereas there are many unnormalized forms. Examples of normalized floating-point numbers are given in Fig. B-3 for two bases of exponentiation. In these examples a 16-bit fraction (including sign bit) and a 7-bit exponent using excess 64 notation are shown. The radix point is to the left of the leftmost fraction bit—that is, to the right of the sign bit.

Figure B-4 shows the formats of the floating-point numbers used on the IBM 370, CDC Cyber 70, and PDP-11 computers.

PROBLEMS

1. About how many decimal digits of precision do each of the formats of Fig. B-4 have?
2. Represent the fraction 27/64 as an IBM 370 short-form floating-point number.
3. What is the decimal value of 41E00000 (hexadecimal)
   a. as an IBM 370 floating-point number?
   b. as a PDP-11 floating-point number?
4. Convert the following numbers to 32-bit floating-point numbers for the IBM 370 and PDP-11 and express your answer in hexadecimal: 1, −192, .125.
5. Add the following IBM 370 floating-point numbers and convert the result to decimal:

   41C80000 (hexadecimal)
   +3FF00000 (hexadecimal)

6. On the Cyber 70, the number 3777400000000000000 (octal) is used as one of the representations of infinity. Express infinity in decimal.
7. If the PDP-11 floating-point numbers had the radix point to the right of the fraction instead of to the left of the fraction, what would be the largest expressible number (in decimal)? Compare it to the maximum number with the radix point at the left of the fraction.
8. What is the smallest normalized number larger than 1 (in decimal) on
   a. the 370?
   b. the PDP-11?
9. A 16-bit binary fraction consists of a binary point followed by 16 binary bits. What are the largest and smallest positive 16-bit binary fractions (excluding 0) in 1's and 2's complement, respectively? Bit 15 is the sign bit.
10. The following binary floating-point numbers consist of a sign bit, an excess 128 expo-

nent, and a 2's complement fraction. Normalize them, assuming exponentiation is to base 2.

a. 0 10000000 000101011000
b. 0 10000111 001000011000
c. 0 01111111 001000000000

11. Write a procedure to simulate normalizing a single-precision floating-point number on the 370. The floating-point number is to be represented by a 32-word integer array, each word containing a 1 or a 0.

12. Repeat Problem 11 using the PDP-11's 32-bit floating-point numbers instead of the 370s. Remember that the PDP-11 uses 2 as the base for exponentiation, not 16 as the 370 does. Furthermore, the high order 1 bit is not stored.

APPENDIX

C BOOLEAN ALGEBRA

In ordinary algebra, a variable can take on any value out of the infinite set of real numbers. A very different kind of algebra, in which variables can only take on values from a finite set containing two elements, is very useful for describing computers. This algebra is called **boolean algebra**, after its discoverer, the English mathematician George Boole (1815–1864).

Just as a bit in memory can have either the value 0 or the value 1, a boolean variable can have the value 0 (or FALSE) or the value 1 (or TRUE). Thus boolean algebra is convenient for describing the binary devices and circuits found in digital computers. In ordinary algebra, there are operators that take one or two operands and produce a result. These operators are called **monadic** and **dyadic**, respectively. The addition, subtraction, multiplication, and division operators are examples of dyadic operators. Absolute value and square root are examples of monadic operators. An operator that takes N operands can be equally well regarded as a function of N variables.

Just as ordinary algebra has operators for its variables, so does boolean algebra. However, the fact that boolean variables can only take on the values 0 and 1 greatly simplifies describing these operators. Consider a dyadic boolean operator with operands P and Q. We will denote this operator as $\doteqdot$ to avoid confusion with the operators of ordinary algebra. There are only four possible combinations of input values:

$0 \doteqdot 0 \qquad 0 \doteqdot 1 \qquad 1 \doteqdot 0 \qquad 1 \doteqdot 1$

This means that the operator is completely specified by giving four values, corresponding to the four possible combinations of the input variables. For example,

"0001" is such a specification if we agree that the four combinations of the input variables are always to be given in the order 00, 01, 10, and 11 as above. The operator "0001" is called the AND operator because the result is 1 if and only if both P AND Q are 1. Symbolically,

0 AND 0 = 0 0 AND 1 = 0 1 AND 0 = 0 1 AND 1 = 1

You should note that specifying an operator by enumerating its value for all possible combinations of its operands is not possible in ordinary algebra because there are infinitely many possible combinations of input values. Finiteness has its virtues. Furthermore, if "0001" defines an operator, so does "0111," in this case, the OR operator, which produces a 1 if either P OR Q OR both are 1. In fact, every one of the four-bit strings from "0000" to "1111" defines a boolean operator. Because there are only 16 different four-bit strings, there are only 16 different dyadic boolean operators. Several are used very often and have special names, as shown in Fig. C-1.

P	0	0	1	1	
Q	0	1	0	1	Name of operator
	0	0	0	0	FALSE
	0	0	0	1	AND
	0	0	1	0	P AND NOT Q
	0	0	1	1	P
	0	1	0	0	Q AND NOT P
	0	1	0	1	Q
	0	1	1	0	EXCLUSIVE OR
	0	1	1	1	INCLUSIVE OR
	1	0	0	0	NOR
	1	0	0	1	EQUIVALENCE
	1	0	1	0	NOT Q
	1	0	1	1	P OR NOT Q
	1	1	0	0	NOT P
	1	1	0	1	IMPLICATION
	1	1	1	0	NAND
	1	1	1	1	TRUE

Fig. C-1. The 16 boolean operators on two variables.

There is only one important monadic boolean operator: NOT. This is defined by

NOT 0 = 1 NOT 1 = 0

In other words, NOT returns the value its operand does "not" have.

Boolean expressions involving more than one operator can be evaluated by simply substituting each possible combination of values for the variables. A table giving the value of a boolean expression for all possible values of the variables is called a **truth table**. To illustrate the use of a truth table, let us evaluate the expression

(P NAND P) NAND (Q NAND Q)

1	2	3	4	5
P	Q	P NAND P	Q NAND Q	(P NAND P) NAND (Q NAND Q)
0	0	1	1	0
0	1	1	0	1
1	0	0	1	1
1	1	0	0	1

Fig. C-2. Using a truth table to evaluate (P NAND P) NAND (Q NAND Q).

The truth table for this expression is shown in Fig. C-2. Each row in the table corresponds to one pair of values for P and Q. The first step in evaluating this expression is to evaluate P NAND P, using Fig. C-1. Column 3 shows P NAND P for the values of P given in column 1. Similarly, Column 4 gives Q NAND Q for the values of Q given in column 2. Finally, columns 3 and 4 are NANDed to give the result, shown in column 5. The result is simply P OR Q. This means that (P NAND P) NAND (Q NAND Q) is equivalent to P OR Q.

Such equivalence relations are of considerable practical importance. Imagine a computer with a hardware NAND unit that took two 32-bit words, P and Q, as input and produced a 32-bit result in which each bit was the NAND of the corresponding input bits. This unit could be used to compute P OR Q as well as P NAND Q by using the formula

P OR Q = (P NAND P) NAND (Q NAND Q)

The first step would be to NAND P with itself and save the result in a temporary variable T1. The next step would be to NAND Q with itself and save the result in another temporary variable T2. The last step would be to NAND T1 with T2 to produce P OR Q.

All 16 dyadic boolean operators can be constructed from repeated use of NAND. Furthermore, NOT X can also be computed using NAND:

NOT X = X NAND X

NAND and NOR are the only two operators from which it is possible to construct all the others by repeated application.

PROBLEMS

1. Which of the following are true?

   a. (a AND b) AND c = a AND (b AND c)
   b. a AND (b OR c) = (a AND b) OR (a AND c)
   c. a OR (b AND c) = (a OR b) AND (a OR c)
   d. (a AND b) OR NOT c = (a AND c) OR (b AND c)

2. A logician drives into a drive-in restaurant and says, "I want a hamburger or a hot dog and french fries." The cook flunked out of sixth grade and doesn't know (or care) whether "and" has precedence over "or". As far as he is concerned, one interpretation is as good as the other. Which of the following cases are possible interpretations of the order? (Note that "or" in English means "exclusive or.")
   a. just a hamburger
   b. just a hot dog
   c. just french fries
   d. a hot dog and french fries
   e. a hamburger and french fries
   f. a hot dog and a hamburger
   g. all three
   h. nothing—the logician goes hungry for being a wiseguy.
3. A missionary is lost in Southern California. He stops at a fork in the road from which two roads emerge. He knows that two motorcycle gangs inhabit the area, one of which always tells the truth and one of which always lies. He wants to know which road leads to Disneyland. What question should he ask?
4. There are 4 boolean functions of a single variable and 16 boolean functions of 2 boolean variables. How many boolean functions of 3 variables are there? Of n variables?
5. Construct a truth table for the function: f(P, Q) = (P AND Q) OR NOT (P AND Q).
6. Use a truth table to show that P = (P AND Q) OR (P AND NOT Q).
7. Construct a truth table for the following function of three variables:

   f(P, Q, R) = (P AND Q AND R) OR (P OR NOT Q)
8. Write a boolean function that computes EXCLUSIVE OR using only AND, OR, and NOT.
9. Simplify the expression NOT (NOT (NOT (NOT P))).

INDEX

A

Absolute load module, 328
Accumulator, 80
Add instruction:
 microprogrammed, 157-160
 PDP-11, 95
Address, 25-27
Addressing, 79-98
 base register, 85
 descriptor, 377-381
 direct, 81-82
 immediate, 80-81
 indexed, 84-85
 indirect, 82-84
 modes, 97-98
 PDP-11, 93-97
 position independent, 95, 336
 register, 82
 stack, 87-93
Address space:
 physical, 252
 virtual, 252
Algorithm, 10
ALU (*see* Arithmetic and logical unit)
Amplitude modulation, 43-44
Anticipatory buffering, 228-229
Applications programmer, 6
Architecture (*see* Computer, architecture)
Arithmetic and logical unit, 146-147
ARPA network, 47-49
Array, 116-119
 processor, 24
ASCII, 34-35
Assembler, 7, 289-302
 tables used, 298
 two-pass, 296
Assembly language:
 characteristics, 289-290
 compared to PL/I, 293
 format, 290-293
 level, 7
Associative law, 414-415
Associative memory, 275-276, 303, 375
Asynchronous transmission, 45-46
Autoindexing, 85

B

B1700, 186, 204-211, 376, 383
Baby feeding algorithms, 257-258
Base register, 85, 336
Batch system, 9
Baud, 43, 45
 relation to bit rate, 43
Best fit algorithm, 272

Binary arithmetic, 421-422
division, 172-175
multiplication, 171-172
Binary number, 415
Binary searching, 305-307
Binding time, 333-336
Bit, 25, 416
map, 225
rate, 43
Blocked record, 227
Boolean, 112-113
algebra, 432-435
Bottom-up design, 349, 352-354
Burroughs B1700 (*see* B1700)
Bus, 41, 143
Byte, 27

C

Cache memory, 263-264
Calling sequence, 125
Carrier, 43
Carry, 422
Catalog, 246
CDC Cyber 70 (*see* Cyber 70)
CDC STAR, 24
Cell, 25-27
size, 27
Central processing unit, 18-24
obsolescence, 50
single chip, 49-50
Channel, 34
Character, 113
Character code, 34
error correcting, 34-38
frequency dependent, 38-41
Checkerboarding, 271-273
Chip, computer on, 49-50
Circular buffer, 230-235
Clock, 144-145
Clustering of hash codes, 312
Collect register transfer, 148-149
Collisions, 312-314
Comments, need, 292
Compaction of memory, 272
Compare instruction, 104-106
Compatibility, leading to stagnation, 385
Compiler, 7, 289
Computer:
architecture, 15
organization, 15
Conditional assembly, 320
Conditional jump instruction, 104-106
Controller, 30
Control store, 162
Conventional machine level, 6, 56-140
examples, 56-70
Conversion between number systems, 416-419
Coroutine, 125, 127-130
CPU (*see* Central processing unit)
Cyber 70, 15, 62-65
addressing, 63
CIO call, 234-235
FET, 229-230
file i/o, 229-235
floating-point numbers, 429
input/output, 110
instructions, 65-66, 76-77, 79, 100, 103, 106
job control language, 280
PPU, 34, 63
registers, 64-65
Cycle, 144
stealing, 67
Cylinder, 30

D

Daisy chain recursion, 107
Data:
movement instructions, 99-100
path, 41-42
representation, 111-119
Deadlock, 249
Deadly embrace, 249
Decimal encoding of numbers, 25
Decoded fields, 176
DEC PDP-11 (*see* PDP-11)
Demand paging, 256
Descriptor, 377-381
base register, 275
segment, 273
Dibit transmission, 43
Digital computer, 1
Direct addressing, 81-82
Directory, 246
Direct register transfer, 148-149
Dirty bit, 261, 269
Disabling interrupts, 135
Disk, 30-31
allocation of file space, 222-226
Displacement, 85
Distance, Hamming, 37-38
Distributed computer, 49-50
Distributed network, 47-49
Distributive law, 415
Division:
algorithm, 172-175
method for hashing, 312

Dope vector, 116-117
Drum, 32
Dyadic instructions, 100-102
Dynamic address translation, 277
Dynamic linking, 336-338
Dynamic relocation, 333-336

E

EBCDIC, 34
ESC, 62-63, 232
Einstein's theory of relativity, 237
Emulation, 363-364
Emulator, 363
Encoded fields, 176
Error-correcting code, 34-38
Estimated entry searching, 307
Event channel, 245
Exception, 370-372
Excess notation, 420
Expanding opcode, 73-75
Exponent, 424

F

FET (*see* File, environment table)
FIFO (*see* First-In, First-Out algorithm)
File, 219
  allocation unit, 222-226
  anticipatory buffering, 228-229
  environment table, 229-230
  implementation, 221-226
  index, 224
  indexed sequential, 227
  organization, 221-226
  random access, 220-221
  relation to virtual memory, 278-279
  sequential, 219-220
Finite precision number, 413-415
First fit algorithm, 272
First-In, First-Out algorithm, 260
Floating point number, 112, 424-431
Flow of control, 119-135
FMS (*see* FORTRAN, monitor system)
FORTRAN:
  monitor system, 8-10
  portability of programs, 365
Forward reference, 296
Fraction, 424
Fragmentation, 262
Frequency distribution for English, 38
Frequency modulation, 44
Full-duplex transmission, 46, 47

G

Gate, 143-144
  microinstruction, 161-162
  sequence, 157-160
Get message, 245
GO TO statement, 119, 166

H

Half-duplex transmission, 46-47
Hamming code, 35-38
Hangover, assembly language as, 295
Hardware, 10
  relation to software, 10-11, 141
Hash coding, 308-316
  collisions, 312-315
  division method, 312
  functions, 308, 312-314
  long names, 310
  performance analysis, 314
  power residue method, 312
  quadratic quotient method, 313
Hexadecimal number, 415
High-level language, 7
High-level machine, 375-385
Hit ratio, 264
Honeywell 6180, 27, 273
Horizontal microprogramming, 177-180
Host computer, 48
Host level, 150
Huffman code, 38-41, 383

I

IBM 360/195, 24
IBM 370, 15, 57-62
  addressing, 57-58
  channel program, 108-109
  DCB, 227-228
  direct access file, 227
  file access methods, 228-229
  file i/o, 226-229
  floating-point numbers, 429
  indexed sequential file, 227
  input/output, 108-109
  instructions, 60-61, 75-79, 99, 103, 107
  interrupts, 134-135
  microinstructions, 194-196
  microprogramming, 190-196
  partitioned file, 226
  registers, 62
  sequential file, 226

IBM 370 (*cont.*):
SVC instruction, 229
virtual memory, 277
VM/370, 364-375
ILC (*see* Instruction, location counter)
ILLIAC IV, 24
Immediate addressing, 80-81
Immediate operand, 80
IMP (*see* Interface message processor)
Increment instruction, 103
Indexing, 84-85
Index register, 84
Indirect addressing, 82-84
Infix notation, 89
Input/output, 33-41, 108-111
Cyber 70, 110
devices, 33
disk, 218
IBM 370, 108-109
instructions, 108-111
PDP-11, 110-111
processor, 33-34
virtual, 218-236
Instruction:
boolean, 101
comparison, 104-106
conditional jump, 104-106
data movement, 99-100
design, 71-72
dyadic, 100-102
execution, 19-24
format, 70-79
location counter, 297
loop, 107-108
monadic, 102-104
no operation, 98
procedure call, 106-107
register, 18-19
rotate, 102-103
set, 21
shift, 102-103
types, 98-111
Integer, 112
Interchange sort, 307-308
Interface message processor, 48-49
Interleaving, 145
Interpretation, 2, 344-346
Interpreter, 2, 20-21
examples, 20, 167-169
job control language, 280
relation to hardware, 21
Interprocess communication, 245-246
Interrupt, 131-135
disabling, 135
nesting, 132
priority, 135
vector, 135
I/O (*see* Input/output)

J

JCL (*see* Job control language)
Job control language, 279-280
Jump, 119

K

Key, 221, 303

L

Language, 3
high-level, 7
machine, 1
problem-oriented, 7
source, 288
target, 288
Least recently used algorithm, 259-260
Level, 3
assembly language, 7, 288-343
conventional machine, 6, 56-140
microprogramming, 5, 141-214
operating system machine, 6, 215-287
relation to virtual machine, 4
Linear searching, 304-305
Linkage editor, 328
Linkage segment, 338
Linked list, 114
Linker, 328-338
Linking, 328-338
dynamic, 336-338
Linking loader, 328
Literal, 298
Loader, 328
Load module, 328
Local storage, 142
Location, 25-27
Logarithmic searching, 305
Logical record, 219
Look ahead, 181
Loop instruction, 107-108
LRU (*see* Least recently used algorithm)

M

Machine:
language, 1
multilevel 4, 10-12, 14, 344-386
virtual, 2-4

Macro, 316-328
call, 317
conditional expansion, 320-322
definition, 316
expansion, 317
implementation, 326-328
nested calls, 322-323
nested definitions, 325-326
parameters, 319-320
processor, 347-349
recursive, 323-325
Magnetic tape, 29-30
Mailbox, 245
Mantissa, 424
MAR (*see* Memory, address register)
Marginal indexing, 117-119
Mask, 101
bit, 135
MBR (*see* Memory, buffer register)
Memory, 24-33
address register, 27-28, 145-146
associative, 275-276, 303, 375
buffer register, 27-28, 145-146
bus width, 145
cycle, 145
cycle ratio, 180-183
hierarchy, 263-264
interleaving, 145
optical, 32
port, 145
read/write, 150
secondary, 29-33
virtual (*see* Virtual memory)
Metabit, 28-29, 377-378
Microinstruction, 148, 161-162
B1700, 209-211
IBM 370, 194-196
PDP-11, 201-204
Micromemory, 162
Microprogram, 5
Microprogram counter, 162
Microprogrammed gate control, 160-161
Microprogramming, 5, 141-142
Microprogramming level, 5, 141-214
B1700, 204-211
design, 175-188
IBM 370, 190-196
PDP-11, 196-204
Middle-out design, 349, 354
Minus zero, 105, 420-421
ML/I, 347-349
Modem, 45
Modulation, 43-45
Monadic instruction, 102-104
MU5, 24, 183
MULTICS:
compared to 370, 277-279
compared to TSS/67, 293
directories, 247
dynamic linking, 338
virtual memory, 273-278
Multilevel machine, 4, 10-12, 14, 344-386
history, 7-10
Multiplexer, 199
Multiplication algorithm, 171-172
Multiprocessor, 22
Multiprogramming system, 217

N

Nand, 147, 433-434
Nanomemory, 183-185
Nanosecond, 144
Negate instruction, 103
Negative binary number, 420-421
Nesting, 132
Network, 47-50, 364
node, 47
reason for existence, 47
research about, 49
topology, 47-48
Nondeterministic process, 352
Nonrestoring division, 175
Normalized floating-point number, 430

O

Object module, 288, 332-333
Object program, 288
Octal number, 415
One's complement, 105, 420
Opcode, 70
Operating system, 6
machine level, 215-287
Organization (*see* Computer, organization)
Orthogonal instruction, 100
OSML instruction, 216
Overflow, 104, 414-415
Overlapped instruction execution, 180-183
Overlay, 250

P

Packed data, 101
Packet, 49
switching network, 47-49
Page, 252
fault, 256

Page (*cont.*):
frame, 253
replacement algorithm, 259-260
size, 262
table, 254
Paging, 250-252
compared to segmentation, 267-268, 273
Parallel instruction execution, 22-24, 180-183
Parallel machine, 22-24
Parallel memory operation, 145-146
Parallel processing, 237-246
simulated, 237-238
true, 237-238
Parameter, 124-125
Parity, 34-35
Pass, 296
Password, 246-247
PC (*see* Program, counter)
PDP-11, 15, 65-70
ADD instruction, 95, 97
addressing, 67-68, 93-97
contiguous file, 235
disk i/o, 218
EMT instruction, 236
file i/o, 235-236
floating-point number, 429
input/output, 110-111
instructions, 69-70, 78, 100, 103, 107
interrupts, 135
linked file, 235
microinstructions, 201-204
microprogramming level, 196-204
MOV instruction, 96
overflow bit, 104
registers, 70
Unibus, 66-67, 199-201
virtual memory, 268-273
Peripheral processing unit, 34, 63
Perplexed children, 173
Phase modulation, 44
Pipeline machine, 24
Pointer, 84
Polish (*see* Reverse Polish)
Portability techniques, 354-364
brute force, 356-357
do-it-yourself virtual machines, 360-363
emulation, 363-364
networks, 364
UNCOL, 357-360
universal programming language, 355-356
Portable programs, 354-364
Position independent addressing, 95
Position independent code, 336
Postfix notation (*see* Reverse Polish)
Power residue method, 313
PPU (*see* Peripheral processing unit)
Prefetching, 181
Priority interrupt, 135
Privacy, 369
Privileged instruction, 370
Problem-oriented language, 7
Procedural extension, 349
Procedure, 120-125
call, 106-107
recursive, 120-126
Process, 12-14
creation, 238
properties, 14
state, 12
state vector, 12-14
synchronization, 242-245
Processor, 14, 18-24
array, 22, 24
pipeline, 23, 24
relation to interpreter, 21
Producer-consumer problem, 239-245
Program, 1
counter, 18
status word, 134
Protection, 246
Protocol, 47
Pseudoinstruction, 290
PSW, 134, 135

Q

Queued file access, 228

R

Race condition, 239-242
Radix, 415
Rat, as i/o device, 33
Record:
logical, 219
physical, 232
zero length, 232
Recursion, 107
Register, 18, 142-143
addressing, 82
transfer, 148-149
Relative error, 427
Relocation:
bits, 333
constant, 330
dictionary, 333
dynamic, 333-336

Relocation (*cont.*):
register, 336
using base registers, 336
Resume, 129
Reverse Polish, 89-93
conversion to infix, 89-90
evaluation, 90-93
Right-adjusted data, 101
Robin Hood and segmentation, 265
Rotate instruction, 102-103
Rounding, 427

S

Scatter register transfer, 148-149
Scratchpad storage, 142
Searching, 303-307
Search key, 303
Secondary memory, 29-33
Sector, 30
Seek, 30
Segment, 266
descriptor, 269, 273-274
fault, 271
Segmentation, 264-268
compared to paging, 267-268, 273
Self-modifying channel program, 372-373
Self-relative addressing, 95
Self-virtualizing machine, 364-375
goals, 367-369
implementation, 370-375
Semaphore, 242-245
Send message, 245
Sequential access, 30
Sequential process (*see* Process)
Shadow page table, 373-375
Shell sort, 307-309
Shift instruction, 102-103
Shift unit, 148-149
Signed magnitude number, 420
Simplex transmission, 46
Skew register transfer, 148-149
S-machine, 376
Sneaky write, 261
Software, 10
relation to hardware, 10-11, 141
Sorting, 307-308
Stack, 87
addressing, 87-93
frame, 121
high-level machine, 382
machine (example), 150-153
macro expansion, 326-328
Stack (*cont.*):
nested interrupts, 132-134
pointer, 87
reverse Polish, 90-93
Star network, 47-48
Start bit, 45
State vector, 12-14
Stop bit, 45
Storage (*see* Memory)
Store and forward network, 47-49
Stride, 116
String, 113-116
Structured computer organization, 1-435
Subcycle, 157
Subroutine, 106
Subscripting, 116-119
Supervisor call, 9, 229
Symbol table, 298
comparison of techniques, 315-316
hash-coded, 308-314
SYN, 35, 45
Synchronization of processes, 242-246
Synchronous transmission, 45-46
Systems programmer, 6

T

Tape, magnetic, 29-30
Target level, 150
Task, 239
Telecommunications, 42-47
Telephone company, as monopoly, 43
Telephone line, 43
Terminal IMP, 48
Test microinstruction, 161-162
Thrashing, 260
Time sharing, 9
self-virtualizing machine, 368-369
TIP (*see* Terminal IMP)
Top-down design, 349-352
Towers of Hanoi, 120-125
Track, 30
Translation, 2, 288, 346-347
Transparency of interrupts, 132
Trap, 130-131
Truth table, 433-434
Tuning, 293-294
Two-dimensional memory, 266
Two-pass assembler, 296
Two's complement, 420

U

UNCOL, 357-360
IBM 370, 358-360

Underflow, 425
Unibus, 66-67, 199-201

V

Vertical microprogramming, 177-180
Virtual address space, 252
Virtual i/o instruction, 218-236
Virtual machine, 2, 3
 fault, 371-372
 exception, 365
 relation to level, 4
Virtual memory, 249-279
 alternative to file i/o, 278-279
 IBM 370, 277
 MULTICS, 273-278
 paged, 250-256
 PDP-11, 268-273
 segmented, 264-268
VM/370, 365-367

W

Wakeup waiting bit, 242
Word, 27
Working set, 258-259